IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received 14 October 2022, accepted 28 October 2022, date of publication 1 November 2022, date of current version 9 November 2022.

Digital Object Identifier 10.1109/ACCESS.2022.3218902

RESEARCH ARTICLE

Fast Witness Generation for Readable GUI Test Scenarios via Generalized Experience Replay

YAVUZ KOROGLU, (Member, IEEE), AND ALPER SEN, (Senior Member, IEEE)
Department of Computer Engineering, Boğaziçi University, 34342 Istanbul, Turkey
Corresponding author: Yavuz Koroglu (yavuz.koroglu@boun.edu.tr)

ABSTRACT Verifying the functional behavior of graphical user interface (GUI) applications is essential for reducing post-release issues. In practice, a developer/tester performs this verification by executing a sequence of GUI actions and then witnessing the expected behavior on the GUI screen. An automated witness generator facilitates the verification process. However, creating an unambiguous and monitorable specification for the correct behavior and then generating the correct GUI actions to trigger that behavior is challenging. In this study, we propose FARLEAD2, an automated witness generator that uses unambiguous, monitorable, and easy-to-read staged test scenarios (STSs) to specify expected behavior. FARLEAD2 maximizes its effectiveness and performance using generalized experienced replay (GER) to exploit the experience gathered from previously witnessed scenarios on new, unwitnessed test scenarios. To the best of our knowledge, STS and GER are novel improvements to GUI testing. Our evaluation of Android GUI applications shows that FARLEAD2 effectively generates a witness 95.7 times out of 100 and does it in 520 seconds, on average, indicating that FARLEAD2 is approximately 65 percent faster and 6.3 percent more effective than its best predecessor.

INDEX TERMS Experience replay, functional testing, GUI testing, reinforcement learning.

I. INTRODUCTION

Graphical user interface (GUI) applications play a huge role in mobile devices. Statistics show that between 2019 and 2021, on average, a hundred thousand new mobile applications have been released on Google Play every month [1]. At this rate, it is inevitable for incorrect behavior to occur in application releases. A survey [2] shows that 78*%sign* of mobile GUI application users regularly encounter bugs causing applications to fail their intended functions. To minimize incorrect behavior without slowing down the software development process, developers need a practical way to verify functionality as much as possible before release.

In practice, a developer/tester verifies the correct behavior by observing it after executing a sequence of GUI actions with the help of a test automation tool. A GUI test automation tool can re-execute a given GUI action sequence, saving the developer/tester from manually executing every GUI action. Furthermore, the developer/tester may insert simple assertions between GUI actions, such as observing a text. Then, these assertions automatically verify correct behavior.

A developer/tester using a test automation tool must still determine GUI actions and assertions related to the GUI function under test. According to a bibliometric analysis [3], automated GUI test generators have been a growing research topic for 30 years. A GUI test generator produces GUI action sequences, saving the developer/tester from manually determining the GUI actions.

In mobile GUI testing studies, many test generators check fatal exceptions (crashes), target structural coverage criteria, or both [4], [5], [6], [7], [8], [9], [10], [11], [12], [13], [14], [15], [16], [17], [18], [19], [20], [21], [22], [23], [24], [25], [26], [27]. Tests produced by these generators excel at detecting crashes and exploring as many GUI screens as possible. However, they still may not witness a GUI function under test because they do not specifically target that GUI function. Some specialized test generators focus on accessibility issues [28], energy bugs [29], common app-agnostic problems [30], [31], and targeting sensitive APIs [32].

The associate editor coordinating the review of this manuscript and approving it for publication was Porfirio Tramontana.

These studies show that a specialized test generator is a necessity to target a specific testing goal.

Most existing research on mobile GUI testing focus on structural coverage and bugs/crashes, so the goal of verifying GUI functions remains a neglected research topic. Our experience with real-world mobile banking applications [33] shows that a state-of-the-art test generator is ineffective in witnessing GUI functions located deep in the GUI application. Hence, a practical test generator must exhibit better performance and effectiveness. In addition, the same experience shows that developers need an unambiguous, readable, maintainable, and automatically monitorable test scenario language to specify example use cases for GUI functions. The test scenarios then would act as automated test oracles for GUI functions.

We propose a novel test generator called fully automated reinforcement learning driven-2 (FARLEAD2). FARLEAD2 aims to maximize its effectiveness and performance via generalized experience replay (GER), a novel technique that exploits the experience gathered from previously witnessed scenarios on new, not-witnessed test scenarios. In addition, it introduces the novel staged test scenario (STS) language. STS is not just unambiguous, readable, and automatically monitorable but is also divided into multiple stages. These stages facilitate FARLEAD2's learning process by using the test scenario as a guide that positively rewards FARLEAD2, as it witnesses intermediate stages instead of just presenting it as a Boolean test oracle.

Fig. 1 presents the overview of FARLEAD2. First, the AUT gets installed on a mobile device. The mobile device can be an emulator, a VirtualBox guest, a physical device, or a finite state transition system modeling the device. The only requirement is that it accepts an action as input and outputs its device state.

Second, the Developer/Tester provides a test scenario in the form of an STS related to the AUT. The STS monitor receives the STS, then computes the monitor state and calculates the immediate reward to supervise the reinforcement learning (RL) agent. Note that FARLEAD2 uses RL, a popular semi-supervised machine learning technique. RL outperforms humans in many fields like resource management [34], traffic light control [35], chess [36], Atari [37], and chemistry [38]. Furthermore, to the best of our knowledge, the best functional witness-generating predecessor of FARLEAD2 for Android uses RL, too [39], [40].

Third, the GER agent generates labels by processing every generalized unit experience residing in the experience database, where a generalized unit experience is a transition between two device states via an action. The GER agent receives rewards from the STS monitor and learns the initial policy. After exhausting all the generalized unit experiences, the GER agent sends the initial policy to the RL agent.

Fourth, after receiving the initial policy, the RL agent searches for a witness (a replayable action sequence consistent with the given STS) by selecting an action according to its current policy and executing it on a mobile device. Then, the RL agent observes the device state and generates a generalized unit experience along with labels, storing the generalized unit experience in the experience database and sending the labels back to the STS monitor. The STS monitor calculates the reward from the labels and sends it back to the RL agent. The RL agent learns from the reward and continues searching until it finds a witness or gives up the search after a predefined limit.

Our main contributions to the literature are

1) the novel GER as a mobile GUI testing improvement for effectiveness and performance,
2) the novel STS as a readable test scenario instead of an LTL specification or Gherkin script, and
3) an experimental evaluation of FARLEAD2, demonstrating its superiority over its best predecessor in functional witness generation for Android GUI applications.

The remainder of this paper is organized as follows. Section II discusses the related work. Section III explains the necessary background on GUI testing, RL agents, and ER. Section IV describes our method. Section V presents our evaluation of FARLEAD2. Section VI discusses threats to validity. Section VII concludes by summarizing our findings and stating future research avenues.

II. RELATED WORK

In this section, we discuss related work in three categories,

1) GUI testing,
2) Android runtime monitoring tools that enable automatic following of a specified test oracle, and
3) Studies combining RL with a formal specification language, such as linear-time temporal logic (LTL) formulae, where an LTL formula acts as an automated oracle.

A. GUI TESTING

Graphical user interface (GUI) testing is the system testing of an application under test (AUT) through a GUI front-end [41]. As such, GUI testing naturally involves verifying the functional behavior of the AUT. This verification task has two challenges: automated test oracles and test generation.

1) AUTOMATED GUI TEST ORACLES

According to a systematic mapping of GUI testing works [41], most studies on GUI testing do not use any test oracle. We consider these articles to be out of our scope because they do not propose fully automated GUI testing methods.

In addition to being a necessity for fully automated testing, a test oracle also contributes significantly to test effectiveness [42]. GUI testing studies with test oracles mostly used either state references or crashes as test oracles [41]. State referencing is the practice of storing GUI states during test execution and then reviewing the stored states to evaluate AUT behavior. State references are impractical, because the number of GUI states available for a GUI application is

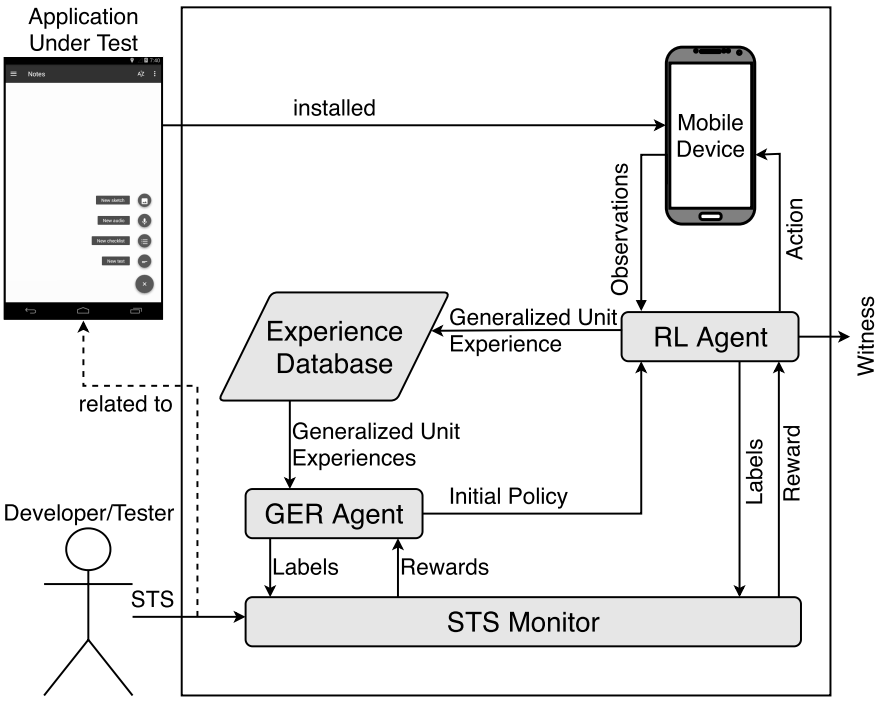


FIGURE 1. FARLEAD2 Overview.

typically large. Therefore, reviewing every GUI state is practically infeasible for the developer/tester.

When the test oracle checks for application crashes instead of state referencing, any crash-free behavior passes the test. Therefore, the test oracle fails to distinguish the correct behavior even if it observes that behavior and accepts faulty behavior if it does not crash. Therefore, the crash-type test oracles are also not particularly useful for functional verification.

Several studies [19], [23], [43] have used structural coverage criteria as a test oracle. However, traditional coverage criteria work poorly for GUI testing because they are suitable for conventional software systems and not for GUI applications [44], [45]. In addition, a test oracle targeting structural coverage suffers from a problem similar to crash-type test oracles; it may ignore correct behavior or accept faulty behavior as long as it observes a structural coverage increase.

Formal methods-based GUI test oracles target the automatic verification of the correct behavior. These test oracles either monitor a model or a specification during test execution and decide whether the test triggered a target behavior during runtime.

Model-based GUI test oracles require an approximately accurate GUI model to evaluate whether a test has passed. However, these models are often unavailable. Therefore, in practice, the test generator crawls the AUT to generate a model. However, if no one specifies the correct behavior, the crawler cannot know what to look for and is unable to generate the necessary model for verifying a target function. Moreover, conventional GUI models such as finite-state machines do not have sufficient descriptive power for verification [46]. Hence, a model-based GUI test oracle is unsuitable for verifying app-specific functional behaviors.

Initial specification-based GUI test oracle studies used operators with first-order logic semantics to describe the test oracle [45], [46]. Subsequently, GUICOP [47], [48] used a custom specification language with variables, properties, constraints, and relational operators based on propositional logic. These relational operators rely on arithmetic comparisons to describe GUI widgets relative to each other, such as a GUI widget residing next to another widget. Finally, our previous works [39], [40] used linear-time temporal logic (LTL) to describe similar test oracles for Android GUIs. In contrast to other logic systems, LTL has the advantage of providing a natural way to express a target behavior over time. Even so, our experience with a real-world banking application [33] shows that any logic-based specification as a test oracle is impractical because the developer/tester'

1) Must become proficient in the language or underlying logic (propositional, first-order, or temporal),
2) Maintains specifications, which would be easier if they were natural language instead of logical expressions, and
3) Presents the test oracle as a test scenario or a software requirement use case to the customer, who does not necessarily know coding or any logic system but needs to know that the AUT's functions work correctly.

Hence, a practical test oracle for verifying functional behavior should be a test scenario close to a natural language but

TABLE 1. Android GUI Test Generators.

Year	#	Name	Goal
N/A	1	Monkey [4]	Crash Detection
2012	2	ACTEve [5]	Structural Coverage
2013	3	A^3E [6]	Structural Coverage
	4	DynoDroid [7]	Coverage & Crash
	5	Orbit [8]	Structural Coverage
	6	SwiftHand [9]	Structural Coverage
2014	7	EvoDroid [10]	Structural Coverage
	8	GreenDroid [29]	Energy
	9	MobiGUITAR [11]	Crash Detection
	10	PUMA [12]	Coverage & Crash
	11	Quantum [30]	Common Bugs
2015	12	MonkeyLab [13]	Structural Coverage
2016	13	CrashScope [14]	Crash Detection
	14	Sapienz [15]	Coverage & Crash
	15	TrimDroid [16]	Structural Coverage
2017	16	DroidBot [32]	Sensitive APIs
	17	Stoat [17]	Structural Coverage
2018	18	CrawlDroid [18]	Crash Detection
	19	MATE [28]	Accesibility
	20	QBE [19]	Coverage & Crash
	21	TCM [20]	Crash Detection
	22	SwiftHand2[a]	Structural Coverage
	23	LAND [21]	Structural Coverage
2019	24	Paraaim [22]	Structural Coverage
	25	FARLEAD-Android [39]	Functional Witness
2020	26	Q-testing [23]	Coverage & Crash
2021	27	GENIE [31]	Common Bugs
	28	DroidBotX [24]	Coverage & Crash
	29	SQDroid [25]	Coverage & Crash
	30	Deep-GUIT [26]	Coverage & Crash
2022	31	SARSA [27]	Coverage & Crash
2022	32	ODIN [49]	Coverage & Common Bugs
2022	33	Fastbot2 [50]	Coverage & Crash

[a] https://github.com/wtchoi/swifthand2

also unambiguous and runtime monitorable. Our experience with readable UI automation syntaxes, such as Gherkin [33], also shows that predefined semantics are not intuitive for such syntaxes, making them impractical for runtime monitoring.

To the best of our knowledge, a readable test scenario language that is also a practical automated GUI test oracle such as the STS language is nonexistent in the literature.

2) AUTOMATED GUI TEST GENERATION

The second challenge in verifying the functional behavior of GUI applications is to generate the correct GUI actions that triggers this behavior. Note that this challenge is different than AI-based planning for GUI testing [45], [51], which produces high-level test cases but leaves the coding of all critical decision points of a low-level, executable test to the developer/tester [44].

Several GUI test generators including AutoBlackTest [52], AntQ [53], TESTAR [54], QBE [19], curiosity-driven Q-testing [23], DroidBotX [24], SQDroid [25], Deep-GUIT [26], and SARSA [27] use RL to guide the test generation. All of these test generators are exploratory tools because they aim to explore a given AUT quickly and as much as possible. Hence, they infer reward functions from the structural coverage criteria. However, verifying a specific GUI function requires a goal-oriented testing tool rather than an exploratory tool.

Table 1 shows the 33 Android GUI test generators investigated in chronological order. Among these test generators, 27 tools target structural coverage, crash detection, or both. Some specialized test generators focus on accessibility issues, energy consumption problems, common bugs, and triggering of sensitive APIs. These studies show that a specialized test generator is necessary to achieve a specific testing goal.

FARLEAD-Android is the first RL-driven GUI test generator that targets user-specified GUI functions by generating witnesses for these functions. Note that SQDroid uses functional semantics to improve structural coverage and does not target specific GUI functions. Our evaluation in Section V shows that FARLEAD-Android is not always effective in witnessing test scenarios that verify functional behavior. When ineffective, FARLEAD-Android executes as many GUI actions as possible on the AUT without producing anything useful for the GUI function under test, throwing away the experience it could exploit. This problem reinforces the FARLEAD-Android's impracticality for the developer/tester.

To the best of our knowledge, FARLEAD2 is the first RL-driven GUI test generator that targets user-specified GUI functions and utilizes previous experience to boost the test generator effectiveness.

B. RUNTIME MONITORING FOR ANDROID

A runtime verification (RV) tool monitors the AUT and reports whether an LTL test oracle passes. RV-Droid [55], RV-Android [56], ADRENALIN-RV [57], and Android-SRV [58] monitor Android AUTs. However, these tools monitor the LTL properties at the source code level. Instead, FARLEAD2 monitor was at the GUI level. In addition, the RV tools do not generate tests whereas FARLEAD2 generates tests while monitoring.

C. RL-LTL STUDIES

Several studies [59], [60], [61], [62] have developed RL-LTL systems. Using RL, these systems learn to obey the constraints specified in LTL. These RL-LTL systems must continuously perform their given tasks, without termination. Hence, as in typical RL-LTL approaches, they must converge to an optimal policy to guarantee the highest reliability. Instead, FARLEAD2 is an RL-LTL system with a finite task; therefore, it can terminate once the task is complete. Therefore, FARLEAD does not have to converge to an optimal policy, thereby saving from the learning time.

III. BACKGROUND

The following two subsections describe (i) the Android GUI environment (ii) the RL agent and (iii) ER.

A. ANDROID GUI ENVIRONMENT

We now discuss GUIs, GUI actions, test scenarios, and reward values.

TABLE 2. Supported GUI Actions.

Action Type	Universal	Related Widget	Parameters
Menu	✔	✖	-
Back	✔	✖	-
2×Back	✔	✖	-
Click	✖	✔	-
Long-Click	✖	✔	-
Scroll-Up	✖	✔	-
Scroll-Down	✖	✔	-
Scroll-Left	✖	✔	-
Scroll-Right	✖	✔	-
Write	✖	✔	text
Reinit	✖	✖	-

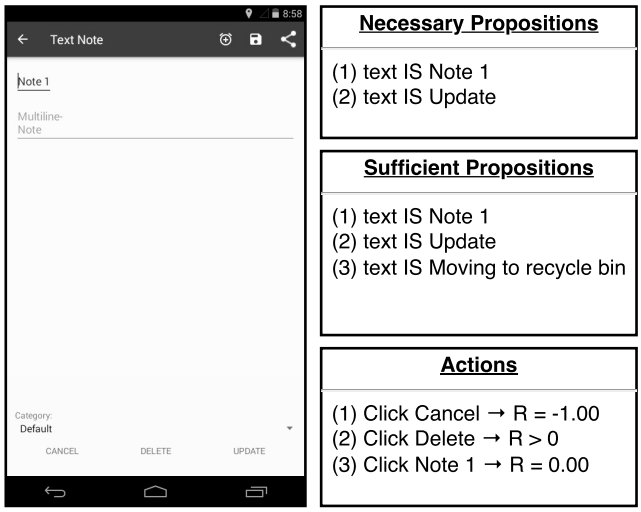


FIGURE 2. An Example GUI Step.

1) GRAPHICAL USER INTERFACE (GUI) AND GUI ACTIONS

A GUI is a visual medium through which a user interacts with the AUT. We aim to automatically verify a GUI function where a GUI function is an operation of the AUT according to the software requirements of the AUT. Typically, witnessing a test scenario ensures the correct behavior of a GUI function, where the test scenario is just an example use case of a GUI function. A test generator witnesses a test scenario by executing a GUI action sequence consistent with the test scenario. Note that not every GUI action sequence corresponds to a test scenario. Therefore, a GUI action sequence is a candidate, and a candidate is a witness only if all its GUI actions are consistent with the test scenario.

Table 2 lists the GUI actions that we support. *Menu*, *back*, and 2×*back* actions are universal actions that are always enabled. *Click*, *long-click*, *scroll-up*, *scroll-down*, *scroll-left*, *scroll-right*, and *write* have related GUI widgets, where a GUI widget is a GUI component visible on the screen. These actions were enabled only if a related GUI widget appeared on the screen. Technically, we determined the set of currently enabled actions by parsing the XML hierarchy of the current GUI layout. Only the *write* action has a parameter, which is the text to write on its related GUI widget. We mainly deduce this parameter from the test scenario, although it is possible to provide a dictionary of generic text inputs. Finally, the *reinit* action is a GUI action with no related GUI widget and is not universal. It is enabled only at the beginning of the GUI action sequence.

2) TEST SCENARIO AND REWARD VALUES

A test scenario is typically an informal description because it should be human-readable for a developer, tester, or customer with no coding skills.The test scenario must be monitored automatically, which is difficult when it is ambiguous. Hence, a test scenario must be a formal description that is unambiguous and runtime monitorable.

Given a test scenario, we aim to find the witness via trial-and-error, generating many candidates before the one consistent with the test scenario. We generated one candidate per episode, where each episode is a finite sequence of steps. We generated and executed a GUI action at each step. To learn after every step, we maintain a reward variable R. Most of the R values are typical for an RL agent. When $R = +1.00$, the candidate at hand (GUI action sequence) is indeed a witness. When $R = -1.00$, the candidate never becomes a witness because of its previous GUI actions. When $R = 0.00$, the candidate does not get closer to or farther from being a witness. In addition to these typical values, atypical partial reward values vary between 0.00 and +1.00. In this case, the candidate is not yet a witness but satisfies some of the conditions of becoming one. In the literature, using such partial reward values is known as Reward Shaping (RS) [63].

At every step, the monitor calculates the reward value automatically by checking the currently monitored propositions at that step for consistency with the test scenario. All of these propositions are Boolean. We observe some of these propositions during one step. All these observed propositions are labels for that step.

A step has two types of proposition: necessary and sufficient. These propositions create three possibilities for each step. (i) Sufficient propositions represent a subset of the labels. In this case, the monitor returns a positive reward, plus one if this is the last step of the given test scenario. (ii) Sufficient propositions are not a subset of the labels, and at least one necessary proposition is not a label. The monitor then returns a minus one reward. (iii) In all other cases, the monitor returns zero. Note that a proposition is related to either the current GUI action or GUI state, where a GUI state is all the GUI widgets' attributes on the screen.

Fig. 2 illustrates an example GUI step. On the screen to the left, every necessary proposition is a label. However, the "text IS Moving to recycle bin" proposition is not. We must generate and execute the correct GUI action, so this proposition also becomes a label, and we get a positive reward for that. In reality, there are many more GUI actions enabled on

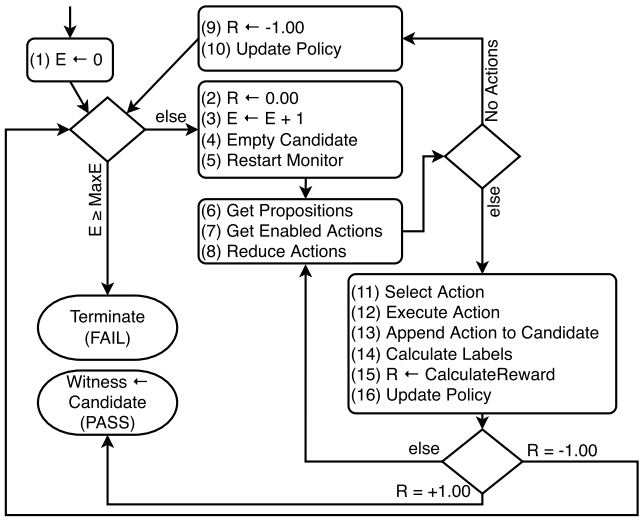


FIGURE 3. RL Agent Flowchart.

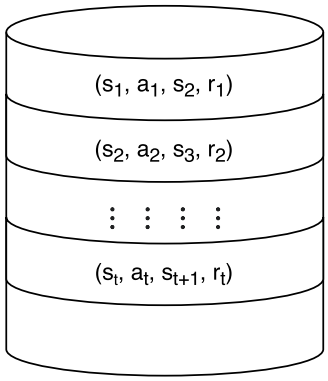


FIGURE 4. Experience Database.

the screen to the left. However, for simplicity, we consider only the *Click Cancel*, *Click Delete*, and *Click Note 1* actions. Clicking *Cancel* closes the current screen. After this action, the monitor generates a minus one reward because (i) the ''text IS Moving to recycle bin'' proposition did not become a label, and (ii) the necessary propositions stopped being labels. *Click Delete* action opens a popup with ''Moving to recycle bin'' text without closing the current screen, making all the propositions labels. Therefore, the monitor generates a positive reward. This reward is plus one if the action witnesses the whole test scenario. *Click Note 1* action clicks the text at the top, so the screen remains unchanged where still, the necessary propositions are labels, but the ''text IS Moving to recycle bin'' proposition did not become a label. Therefore, the monitor generates a zero reward. Note that all propositions in the example are state propositions. Action propositions may constrain what actions a test generator should take. In that case, we automatically reduce the set of enabled actions to avoid any future negative rewards and pursue positive rewards.

B. REINFORCEMENT LEARNING AGENT

Fig. 3 shows the flow of the RL agent. First, line (1) initializes the number of episodes (E) to zero. If the number of episodes is equal to or larger than the predefined maximum number of episodes (MaxE), the RL agent terminates because it has failed to generate a witness. Otherwise, the RL agent starts a new episode through lines (2)–(5). line (2) initializes R as zero. line (3) increments the number of episodes. line (4) empties the candidate. line (5) restarts the monitor, so the monitor starts to observe the test scenario from the beginning. The RL agent begins a new step in lines (6)–(8). Line (6) provides the monitored propositions in the current step. Line (7) obtains the set of enabled actions by the AUT. Line (8) reduces the set of enabled actions according to necessary and sufficient propositions. The RL agent reaches a dead end only if there are no enabled actions after the reduction. In this case, line (9) sets R to minus one, and line (10) updates the action selection policy according to R. Then, the RL agent starts a new episode. Otherwise, line (11) selects a GUI action according to the action selection Policy. Line (12) executes this GUI action, and line (13) appends it to the candidate. line (14) calculates the labels. Finally, line (15) calculates R from these labels, and line (16) updates the action generation Policy. If $R = +1.00$, the candidate is a witness, and the RL agent terminates. If $R = -1.00$, it starts a new episode. Otherwise, the RL agent proceeds to generate a new step by going to line (6). The RL agent eventually terminates because its monitor has an internal step counter that produces a negative one reward if there are too many steps in the episode.

The RL agent effectively generates a witness only if it produces the witness within the predefined maximum number of episodes. Otherwise, there are two possible explanations. Either we need more episodes to find a witness, or the GUI function is nonexistent in the AUT. Note that the RL agent cannot prove the nonexistence of a GUI function such as a model checker.

In addition to verifying GUI functions, we can also reproduce a known GUI bug using a test scenario for the bug. From a test generator's perspective, a GUI bug is equivalent to a GUI function because it reproduces a GUI bug in the same way that it verifies a GUI function.

C. EXPERIENCE REPLAY (ER)

Experience replay (ER) [64] is an improvement in RL. ER exploits the experience stored in an experience database from previous tasks instead of discarding it. Fig. 4 illustrates an experience database as an ordered collection of unit experiences. A unit experience is a quadruple ((s, m), a, (s', m'), r), meaning that the AUT goes from one device-monitor state-pair (s, m) to the next (s', m') by executing action a and obtaining reward r. The main idea is not to throw away but to save all the unit experiences from previous tasks and then reuse all the unit experiences for the task at hand.

One strength of ER is its ability to influence the RL agent's initial policy without executing any actions on the device.

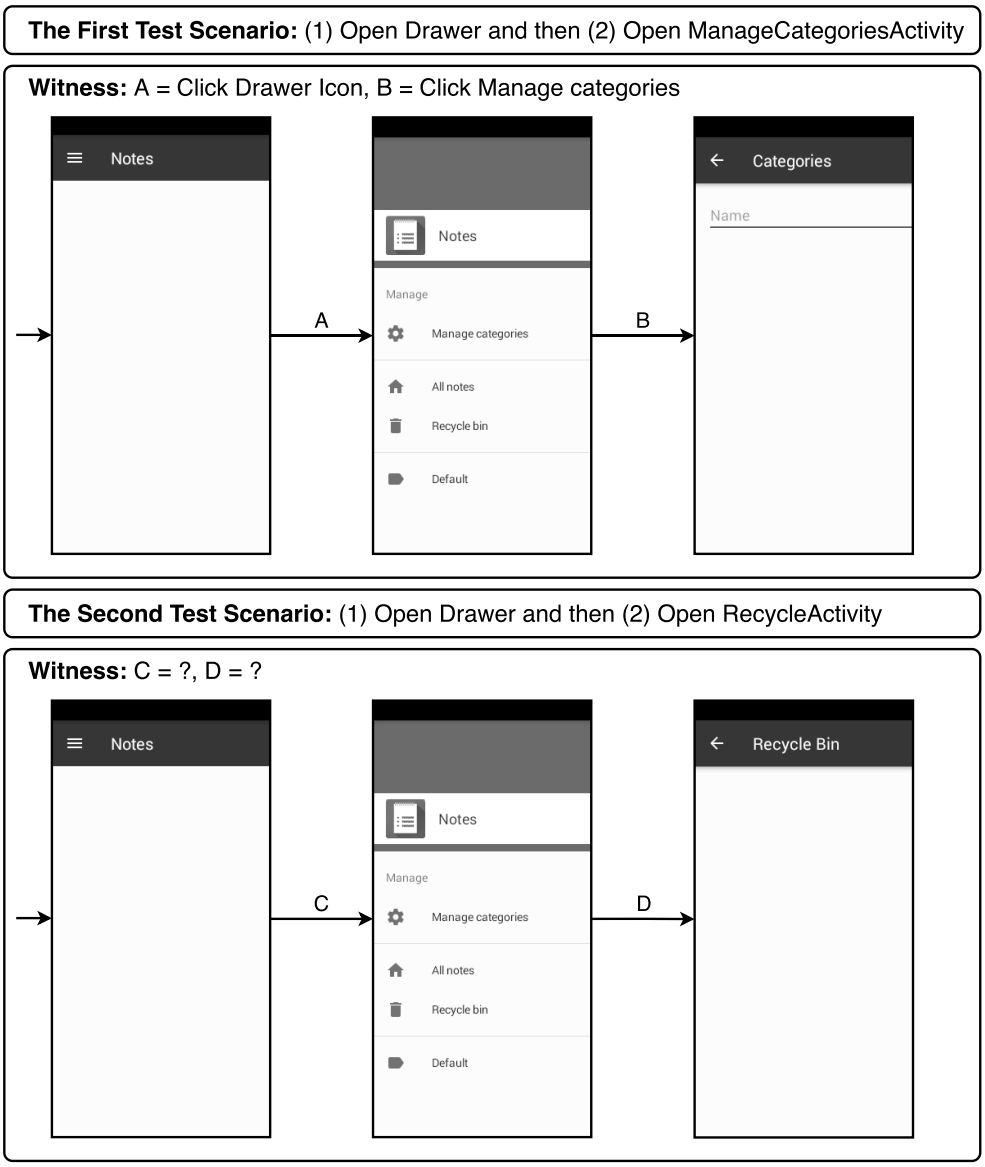


FIGURE 5. Generalized Experience Replay (GER) Example.

Instead, ER immediately provides reward values to the RL agent from its experiences, allowing the RL agent to start with an initially better action generation policy than a random one. As a result, the RL agent should converge towards its objective faster while avoiding the execution costs of all the unit experiences. As ER gathers more unit experiences, it should become faster.

The underlying assumption of ER is generalization via connectionism. Connectionism implies that the unit experiences are related to the task at hand. Otherwise, they can hinder the learning effectiveness and performance instead of enhancing it. Generalization implies that a generalizable pattern exists between unit experiences. Otherwise, it would not be possible to learning anything from the unit experience.

IV. METHOD

FARLEAD2 improves RL-driven witness generation through GER and STSs. This section first explains why the traditional ER would hamper FARLEAD2 performance, so we implement GER. We then discuss STSs in detail.

A. GENERALIZED EXPERIENCE REPLAY (GER)

Every unit experience ((s,m), a, (s',m'), r) in a traditional ER has a fixed reward and monitor states; r, m, and m'. However, once FARLEAD2 witnesses a test scenario, the developer/tester will not use FARLEAD2 again but the already existing replayable witness instead. Hence, the developer/tester will always use FARLEAD2 with unique test scenarios (scenarios that FARLEAD2 has never witnessed before). Every unique test scenario yields a different reward

function and monitor states. Therefore, if FARLEAD2 uses traditional ER, some of the recorded rewards are bound to be misleading for the new test scenario, hampering witness generation's effectiveness and performance.

The STS monitor generates its state and a reward value at every step by checking the labels of that step. The RL agent determines these labels by looking at the step's GUI action a and the device state s' that is reached after executing the GUI action. In other words, the reward r and the monitor state m are functions of the GUI action a and the device state s'. Hence, storing only the device states and GUI actions is sufficient to compute the monitor states and calculate the reward values for any test scenario.

A generalized unit experience is a triple (s, a, s'), meaning that the AUT goes from device state s to device state s' by executing action a. Note that storing only (a, s') is sufficient to calculate the reward value. However, it is insufficient to determine which state-action pair (s, a) that value refers.

Fig. 5 demonstrates an example in which the generalized experience gathered in the first test scenario facilitates witnessing the second. These scenarios involve reaching different AUT screens in two steps. We already have a witness for the first test scenario. This witness has two GUI actions, A and B. For the second test scenario, we do not yet have a witness. Therefore, GUI actions C and D are unknown. Before any exploration, the GER module replays the generalized experience gathered from the first witness. During replay, GUI action A gets a positive reward value because it is consistent with the first step of the second test scenario. However, the GER module assigns a negative reward value to the GUI action B because it is inconsistent with the second step of the test scenario. Consequently, FARLEAD2 selects C=A with no exploration and eliminates B as a candidate for the second step. Overall, the search space for the second witness shrinks, thereby facilitating witness generation.

The GER agent first traverses all the generalized unit experiences in chronological order and receives rewards from the STS monitor. At this point, the GER agent does not learn the initial policy. Instead, it sorts all the generalized unit experiences according to their rewards, in ascending order. Then, it traverses all the generalized unit experiences in this order, learning from each generalized unit experience $\lfloor r+2 \rfloor$ times, where r is the reward value. Overall, re-learning the positively rewarded generalized unit experiences many times and learning them after other experiences significantly increases their impact on the initial policy.

B. STAGED TEST SCENARIO (STS)

Fig. 6 shows an overview of a an STS, where the inside angle and square brackets are variables and optional constructs, respectively. Keywords separated by slashes are alternatives. The main idea behind STS is that it divides the scenario into consecutive STS stages. FARLEAD2 searches for a candidate that witnesses the STS stages in the given order. The developer or tester may specify the maximum number of steps or the time available for the entire scenario or any STS stage.

```
SCENARIO: <ScenarioName>
    [MAXSTEPS = <MaxSteps>]
    [MAXTIME = <MaxTime>]
    STAGE: <StageName>
        [MAXSTEPS = <MaxSteps>]
        [MAXTIME = <MaxTime>]
        INVARIANTS:
            [<Proposition>[AND/OR <Proposition>]...]
            [<Proposition>[AND/OR <Proposition>]...]
            ...
        [STEPS:
            <StepProposition>
            [<StepProposition>]
            ...
        ]
        EVENTUALLY:
            <Proposition>[AND/OR <Proposition>]...
            [<Proposition>[AND/OR <Proposition>]]...
            ...
    [STAGE: <StageName>
        ...
    ]
```

Example Propositions:

```
P = activity CONTAINS MainActivity
Q = text IS Note 1
R = text IS NOT Note 1
S = ActionType IS Write
    AND ActionDetail IS username
    AND ActionParam IS Yavuz
    (Write Yavuz TO username)
```

FIGURE 6. Staged Test Scenario (STS) Overview.

Within the given time constraints and step bounds, a candidate witnesses a stage only if all of its invariants are true (labels) until all of its eventual conditions become true (labels). In other words, all invariants and all eventual conditions are necessary and sufficient propositions, respectively. Note that this is like the until operator in LTL. All the invariants and eventual conditions are Boolean propositions. These propositions do not have to be atomic. They may have several terms connected via AND or OR operators.

We assume that every Boolean proposition is a triple (property, relation, and value). There are two types of properties: action and state properties. Three types of action properties exist. These are ActionType, ActionParam, and ActionDetail, which are linked to the GUI action type, the action parameter, and an attribute of the related widget, respectively. State properties are either crashed, package, activity, or one of any GUI widget's attributes on the screen. The relation is IS, IS NOT, CONTAINS, or NOT CONTAINS.

In Fig. 6, P, Q, R, and S are some example propositions. These propositions become labels only if the current activity name contains MainActivity, the screen has a text that writes exactly "Note 1", there are no texts that write exactly "Note 1", and the GUI action types the word "Yavuz" to username, respectively.

A stage may have optional steps. These steps are a list of action properties. For example, S is a step proposition, and P, Q, and R are not shown in Fig. 6.

The step propositions can be cumbersome to specify. To address this issue, we have developed shorthand notation. FARLEAD2 automatically converts every shorthand notation in the STS into a proper step proposition. Fig. 6 gives the shorthand notation for S between the round brackets.

The initial state of the STS monitor m indicates that the candidate has not yet witnessed any STS stage. Once the STS stage is complete, the STS monitor moves to the next state m'. Reporting the monitor state is essential for teaching the RL agent the correct order of test steps necessary to witness a given test scenario.

V. EVALUATION

This section describes the experimental setup, discusses the research questions, and evaluates the experimental results.

A. EXPERIMENTAL SETUP

1) TEST GENERATORS

In this study, we compared three test generators, random (RND), reinforcement learning (RL), and generalized experience replay assisted RL (GER).

RND generates random GUI actions, ignoring all rewards. We included RND in our experiments as a baseline for evaluation. Therefore, any other test generator should outperform RND.

RL is equivalent to FARLEAD-Android. To the best of our knowledge, FARLEAD-Android is the most effective test generator for producing functional witnesses. Therefore, we aim to improve on RL.

GER is the proposed method for FARLEAD2. Our experiments demonstrate that GER is more effective than and outperforms both RND and RL.

2) EFFECTIVENESS

In our evaluation, a test generator was required to produce a witness within 100 episodes. Therefore, the effectiveness of the test generator is the percentage of times it is successful within this limit. We executed the same test generator ten times for the same test scenario under the same conditions. Thus, the witness generator is a hundred percent effective if it generates a witness ten times. Conversely, it is zero percent effective if it fails at all times. We took the average across all test scenarios to measure the overall effectiveness of the test generator.

In summary, the effectiveness is the expected number of witnesses that the test generator produces out of 100 attempts. An ineffective test generator would waste the developer's time without generating any witnesses; therefore, the most critical aspect of a practical test generator is to be as close to a hundred percent effective as possible.

3) PERFORMANCE

A witness generator outperforms the others if it terminates faster. We have two measures reflecting performance, (i) the total number of steps and (ii) the total seconds it takes until termination. We examine the first measurement to ensure that the latter does not suffer from noise caused by the varying execution times of individual GUI actions on the mobile device. Because we executed the same scenario under the same conditions ten times, we took the average of both performance measures.

4) THE MOBILE DEVICE

Throughout our experiments, the mobile device was a VirtualBox guest with 1024 megabytes of random access memory. The operating system of this device is the Intel x86 port of Android 6.0. Using a VirtualBox guest, we create exact clones of our experimental environment, allowing mass witness generation for different test scenarios in parallel. Furthermore, no physical mobile devices or hardware preparations were required to replicate the experiments.

5) ANDROID APPLICATIONS

We used the Themis Automated Android GUI Testing Benchmark [65] and the F-Droid repository to locate Android applications to evaluate our test generators. Themis is a well-known and maintained benchmark, recently developed to compare Android GUI test generators. F-Droid is an open-source Android GUI application repository. Multiple Android GUI test generators in the literature [19], [24] use Android applications from this repository.

We evaluate the experimental test generators over two Android applications: *Notes* from F-Droid and *Wikimedia Commons* from Themis. *Notes* is a small-sized (2 MBs) Android application similar to the other small note-taking applications in the Themis benchmark, such as *Omni-Notes* and *Scarlet-Notes*. The *Commons* application is a medium-sized (17 MBs) Android application.

The *Notes* application allows users to create four types of notes: audio, text, sketch, and checklist. Furthermore, users can construct categories and divide notes into these categories. This application has a known bug in its sketch notes where the color palette has no black color, preventing users from making black drawings [66]. The *Commons* application allows users to search for pictures in the public domain. Users may view these pictures and their descriptions.

6) TEST SCENARIOS

Our experimental setup has 24 test scenarios: 17 and 7 for the *Notes* and the *Commons* applications, respectively. The complexities of these test scenarios vary between 2 and 13, where we define the test scenario's complexity as the length of its shortest witness. The shortest witness length is the minimum number of steps (GUI actions) necessary to witness a test scenario. We argue that a test generator would face more difficulties in a complex test scenario because of the number of unknown steps that need to be discovered.

Fig. 7 shows an example of a manually generated witness for test scenario 014. The existence of this witness places an upper bound of seven on the complexity of this test scenario.

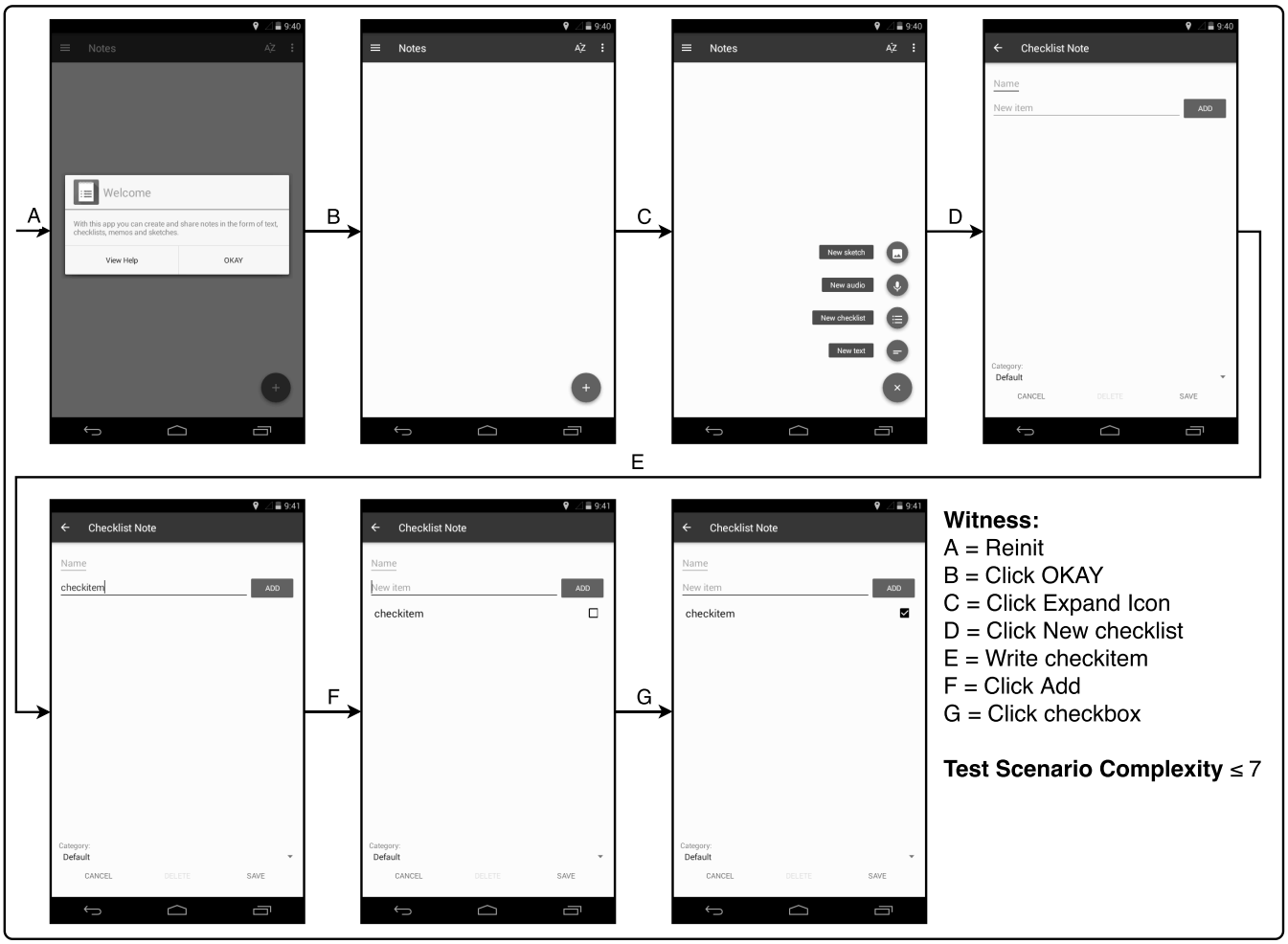


FIGURE 7. A Witness for Test Scenario 014.

We manually produced witnesses for all the test scenarios to determine their complexity.

An STS is a flexible structure, allowing the developer/tester to incorporate apriori information about the test scenario. We call this information hints. According to the hints given in an STS, there are two extreme STS types: declarative and imperative.

A declarative STS only contains the information necessary for a scenario. This information includes (i) the invariants and (ii) the eventual conditions of every stage. Therefore, the developer/tester declares only what the test generator should witness. In contrast, an imperative STS defines the steps of every stage. It shrinks the search space; therefore, there is often only one candidate. However, it is cumbersome to maintain an imperative STS because it requires restructuring after almost any software update, whereas a declarative STS should work across multiple versions of the AUT.

For every experimental test scenario, we have four STSs, with four levels of hints: L4 (imperative), L3 (manual), L2 (automated), and L1 (declarative). Hence, for the 24 test scenarios, we obtained a total of 96 STSs.

Fig. 8 shows the L1-L4 STSs for test scenario 014. The first stage of L1 has no invariants but only one eventual condition, starting the AUT package on the device. The second stage has one invariant, indicating that the AUT package must be active until the second stage's eventual condition is satisfied, so ChecklistNoteActivity is on the screen. Again, the third stage has the same invariant, describing that the AUT package must be active, but now it is until the device ends up in the ChecklistNoteActivity, while there is a text that writes "checkitem" and there is a checked checkbox on the screen. Overall, L1-STS states that (i) eventually, the AUT must be opened. (ii) After that, eventually, the ChecklistNoteActivity must be opened. (iii) Finally, the ChecklistNoteActivity must be on the screen with the checklist containing a checked item, and the "checkitem" text appears on the screen. Whenever FARLEAD2 encounters a text proposition, it automatically considers writing that text to any appropriate GUI widget an enabled action.

We automatically generated L2-STSs from L1-STSs through intent-resolution analysis [6]. Intent resolution analysis is a static analysis of an Android GUI application binary

FIGURE 8. L1-L4 STSs for Test Scenario 014.

that extracts the static activity transition graph (SATG) of an AUT. The SATG determines from any activity to which a tester can go, and using the SATG, FARLEAD2 updates the given STS with extra invariants and eventual conditions. For test scenario 014, FARLEAD2 determined that it could reach ChecklistNoteActivity via MainActivity. Therefore, it automatically restricts its search for these activities by adding activity constraints to the appropriate stages of the STS. Overall, L2-STS reduces the search space without manual effort.

The L3-STS incorporates any hints that the developer or tester may provide, except for the steps (GUI actions) themselves. First, MAXSTEPS defines the maximum number of steps allowed for each stage. Setting every MAXSTEPS condition to an absolute minimum forces the test generator to produce the shortest witness. As a result, the test generator spends more time than finding an arbitrary witness. Therefore, we slightly relaxed MAXSTEPS values. Second, ''actionType'' propositions restrict the action type. Third, an extra stage called ExpandDrawer states that the text ''New'' must appear on the screen before reaching the ChecklistNoteActivity. All these additions shrink the search space but require additional manual effort.

The L4-STS is imperative and describes all the steps, so almost no learning is required. Fig. 7 shows that after the GUI action D, FARLEAD2 must still learn the correct

GUI widget to write on. With the L4-STS, search space is the smallest, but the developer/tester determines all the GUI actions manually, making the manual effort of writing an L4-STS the highest among all STSs. Still, writing an executable test script requires coding skills, whereas an L4-STS is a no-code script. Thus, writing an L4-STS requires less effort than writing a test script.

We have designed one test scenario (four STSs) to reach each activity of the *Notes* application (test scenarios 001-009). Hence, witnessing all the test scenarios achieved full activity coverage. We have created a test scenario (test scenario 012) to reproduce the palette bug [66]. Finally, the rest of the test scenarios concern the main GUI functions of the *Notes* application, namely, creating, deleting, recycling, and categorizing notes. For the *Commons* application, the first five scenarios were activity-reachability scenarios. The remaining two verified different GUI functions of the application.

7) GENERALIZED EXPERIENCE REPLAY (GER) SETUP

GER depends on the experience gathered so far. Our experimental setup starts with no experience and executes GER on the test scenarios in the order of increasing test complexity. Even though GER re-witnesses an STS multiple times in our experiments, it never uses the experience of the same STS. GER selects one run per previous STS and uses the cumulative experience gathered only from these STSs. Hence, we expect GER to produce results similar to those of RL in test scenario 001. GER will have the most experience when it witnesses the most complex test scenario (Notes, test scenario 017). Note that we used separate experience databases for each STS level. Finally, although we perform our experiments in parallel, GER waits for the previous scenarios to finish before generating witnesses.

8) OVERALL

Our experimental setup has three test generators (RND, RL, and GER), 96 STSs, and ten runs for each test generator-STS combination. Hence, there were a total of 3840 experimental runs. We collected four values for every experimental run: (i) success/fail, (ii) total number of steps, (iii) total number of seconds, and (iv) witness length. The first value measures effectiveness, the second and third values measure performance, and the last value measures test complexity.

B. RESEARCH QUESTIONS

Our experimental setup aims to answer the following research questions.

- **RQ1.** (Feasibility) Are all the experimental test scenarios witnessable?
- **RQ2.** (Effectiveness) How much more effective is GER than RL and RND?
- **RQ3.** (Performance) How much performance increase does GER have compared to RL and RND?
- **RQ4.** (Hints) What is the impact of hints (L1-L4 STSs) on the test generation effectiveness and performance?

TABLE 3. Overall Results.

		RND	RL	GER
Overall 96 STSs	**Effectiveness (%)**	67.1	89.4	**95.7**
	# Steps	210	213	**140**
	Time (s)	1102	859	**520**
	Witness Length	6.19	6.69	6.78
Notes 68 STSs	**Effectiveness (%)**	66.5	87.1	94.3
	# Steps	190	238	167
	Time (s)	650	778	555
	Witness Length	5.42	6.16	6.45
Commons 28 STSs	**Effectiveness (%)**	68.6	95.0	99.3
	# Steps	259	153	73.6
	Time (s)	2199	1055	433
	Witness Length	8.01	7.87	7.52

RQ1 verifies that every experimental test scenario has positive utility in evaluating effectiveness, performance, and test complexity. If the underlying GUI function that a test scenario exploits is nonexistent in the AUT, there are no witnesses for that test scenario. Then, the effectiveness will be zero percent regardless of the test generator, and the performance and witness length measurements would be infeasible. We aim to show that at least one witness exists for every experimental test scenario.

RQ2 evaluates the most crucial criterion for a witness generator: its effectiveness. Depending on the test scenario, an ineffective test generator frequently fails in practice, frustrating the developer/tester. We aim to ensure that GER is more effective than RND and RL in witness generation.

RQ3 evaluates how fast a test generator terminates. A faster and more effective test generator would produce more witnesses within a constant testing budget, providing the developer/tester more utility. We aim to demonstrate that GER outperformed RND and RL in our experiments.

Finally, RQ4 aims to determine GER's effectiveness and performance under different STS levels.

C. EXPERIMENTAL RESULTS

Table 3 shows our overall experimental results comparing the effectiveness, time, steps, and witness lengths of RND, RL, and GER. Each result for the *Notes* and *Commons* applications is an average across the STSs of that application. Each overall result is an average across all STSs.

(RQ2 and RQ3) Overall, FARLEAD2 (GER) generated a witness 95.7 times out of 100 and did it in 140 steps and 520 seconds, on average. FARLEAD2 (GER) was 6.3 percent more effective and 339 seconds (approximately 65 percent) faster than its best predecessor, FARLEAD-Android (RL). Furthermore, it was 28.6 percent more effective and 582 seconds (112 percent) quicker than RND.

For the small-sized *Notes* application, FARLEAD2 (GER) generated a witness 94.3 times out of 100 and did it in 167 steps and 555 seconds, on average. Hence, in the *Notes* application, FARLEAD2 (GER) was 7.2 percent more

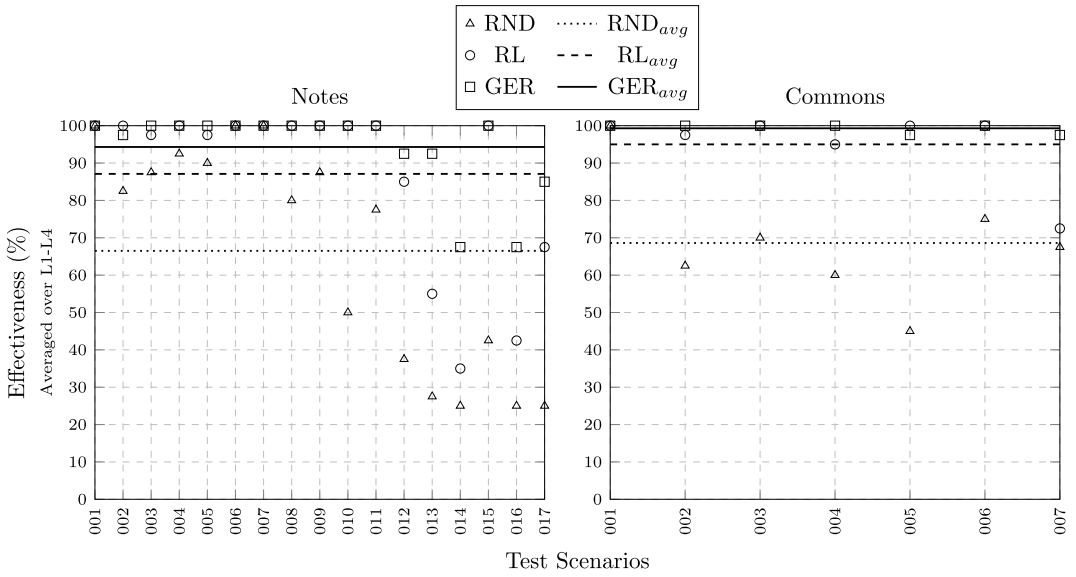


FIGURE 9. Overall Effectiveness Results.

effective and 223 seconds (approximately 40 percent) faster than its best predecessor, FARLEAD-Android (RL). Furthermore, it was 27.8 percent more effective and 95 seconds (17 percent) quicker than RND.

For the medium-sized *Commons* application, FARLEAD2 (GER) generated a witness 99.3 times out of 100 and did it in 73.6 steps and 433 seconds, on average. Hence, in the *Commons* application, FARLEAD2 (GER) was 4.3 percent more effective and 622 seconds (approximately 144 percent) faster than its best predecessor, FARLEAD-Android (RL). Furthermore, it was 30.7 percent more effective and 1766 seconds (408 percent) quicker than RND.

All results in Table 3 show that FARLEAD2 (GER) is consistently more effective and faster than its alternatives, revealing the benefits of experience replay.

Fig. 9 shows the effectiveness scores of RND, RL, and GER for every test scenario averaged across all STS levels (L1-L4). The lines in this figure represent the effectiveness scores listed in Table 3. *(RQ1) RND, RL, and GER did not have zero effectiveness in any test scenario, indicating that all the test scenarios are feasible.*

Fig. 10 shows the test generation times (Fig. 10a) and the number of steps (Fig. 10b) of GER under the L1-L4 STSs. Because there were no discrepancies between step count and time results, we assume the noise in test generation times was negligible. Therefore, in this case, the test generation time is an accurate performance measure.

(RQ4) Fig. 10a shows that for both the Notes and Commons applications, more hints yielded faster test generation times. Without hints (L1 - declarative), it took FARLEAD2 (GER) more than 1000 seconds to generate a witness on average. Enabling L2 STSs significantly improved the average test generation performance. However, Fig. 10a further shows that test scenarios 014 and 016 of the Notes application required more than 4000 seconds with L1 and L2 STSs, respectively, whereas L3 STSs never took more than 2000 seconds. Therefore, with manually generated L3 STSs, GER provided more reliable performance than L1 or L2 STSs. L4 STSs required almost no time to witness.

VI. DISCUSSION

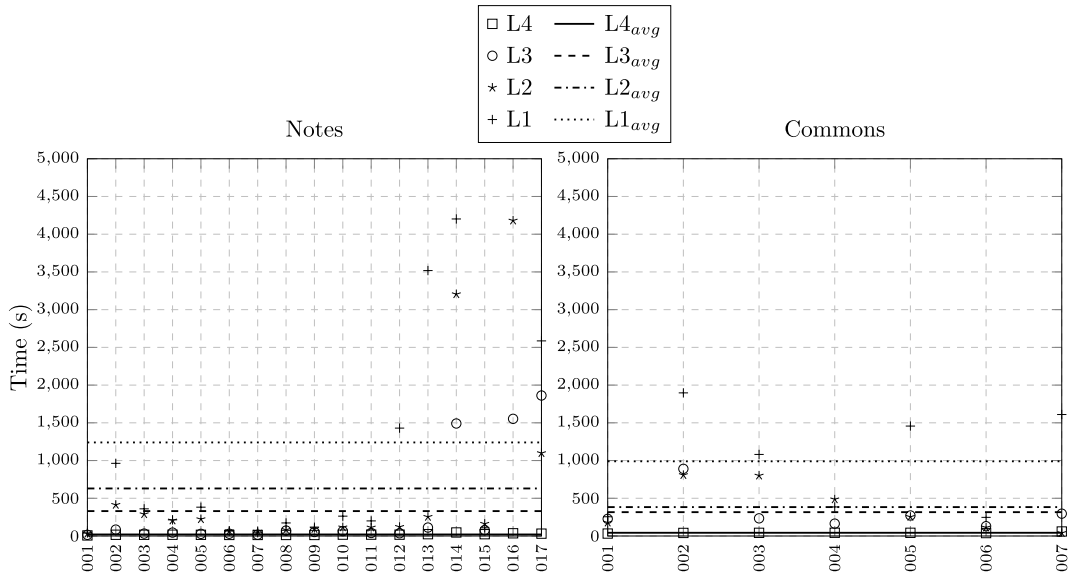
Now, we discuss the threats to the validity of our experimental setup, methodology, and implementation. Specifically, we elaborate on the construct and external validities.

A. CONSTRUCT VALIDITY

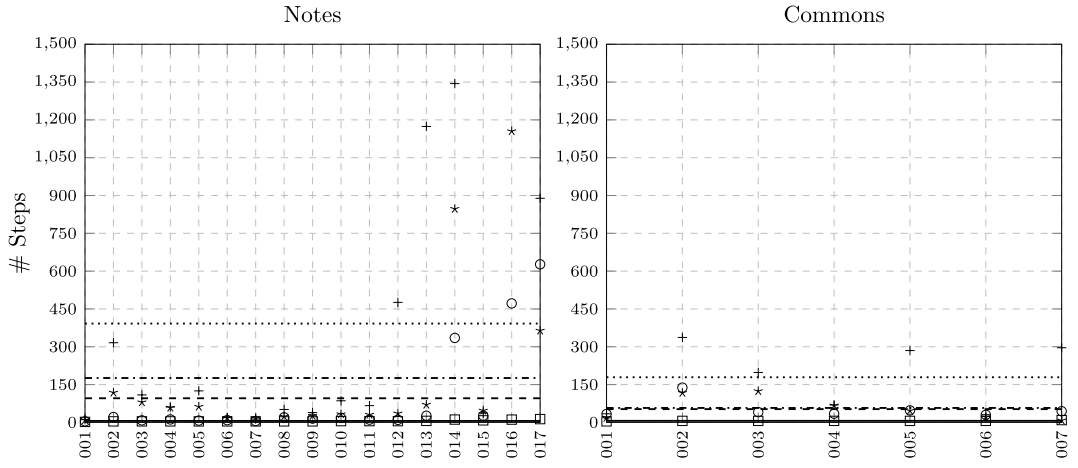
The maximum number of steps allowed in each episode was 30. The highest test complexity in our experiments is less than 30; therefore, this limit will not cause ineffectiveness in any test generator. In addition, our experience with Android GUI applications shows that developers implement simple GUI functions such that any user can perform them via fewer than 30 consecutive actions.

In our experiments, the episode limit is 100. Our experience shows that the test generators become ineffective when this limit is too low. A high episode limit causes the test generator to waste more time when a witness cannot be found. We refrained from finding the optimal episode limit to avoid bias toward our experimental AUTs. Instead, we arbitrarily choose the episode limit, considering only the testing budget.

Because we re-executed every experimental run ten times, the GER setup collected experience for the same scenario multiple times. However, in practice, the developer/tester would never re-execute FARLEAD2 on a test scenario with an already available witness. To reflect this fact in our experimental environment, we forced every run of a test



(a) The Effect of STS Levels on Test Generation Times



(b) The Effect of STS Levels on the Number of Steps

FIGURE 10. The Effect of STS Levels on GER Performance.

scenario to use only the experience gathered during the respective runs of the previous test scenarios. Note that all of our experimental test scenarios are distinct; therefore, GER always uses the experiences of witnessed test scenarios on unwitnessed test scenarios.

Every test generator that we evaluated must monitor the STSs. The original FARLEAD-Android monitors LTLs instead of STSs; therefore, we had to make technical modifications to implement RL in our experiments. During the experiments, we noticed that cycles in the state transitions may create a positive feedback loop that stuck FARLEAD (RL) and FARLEAD2 (GER). Thus, our modified implementation proceeds to the next episode when it encounters a cycle that is not a self-loop. For fairness, we also force RND to go to the next episode in the same cases.

Finally, although there are many Android GUI test generators in the literature, we did not include all of these generators in our evaluation. First, to the best of our knowledge, none of these test generators implement STS monitoring. Second, even if we implemented STS monitoring on top of these test generators, it would be an unfair comparison because these modified test generators cannot benefit from the rewards produced by the STS monitor. Hence, we compared our proposed method to FARLEAD-Android, the best predecessor of FARLEAD2 for generating witnesses for given GUI test scenarios in Android. Note that FARLEAD2 implements experience replay on top of FARLEAD-Android. Therefore, we compared FARLEAD2 with FARLEAD-Android to demonstrate the benefits of experience replay.

B. EXTERNAL VALIDITY

FARLEAD2 (GER) achieved 100 percent activity coverage on Notes and Commons applications. RND and FARLEAD-Android (RL) also reached 100 percent activity coverage in our experiments because we specified one test scenario per activity, and all the test generators producd witnesses for every test scenario. As a result, we did not include activity coverage in our comparisons because it did not distinguish test generator effectiveness and performance when witnessing a test scenario. The developer/tester must provide feasible test scenarios for all activities of an AUT to achieve 100 percent coverage.

The set of supported actions directly affects witness generation effectiveness. Our experience shows that one *back* is sometimes insufficient to return to the previous activity. Hence, we introduced the $2\times back$ action. In addition, some real-world applications involve dynamically-loaded activities. These activities may require the user to wait for a few seconds before performing any action. Therefore, for practical use of FARLEAD2, it may be necessary to introduce the option of waiting for a few seconds as a GUI action.

The Android applications in our experiments were standalone. However, for example, the developer/tester needs at least two devices to test a messaging application. Thus, the witness becomes not just a GUI action sequence but at least two GUI action sequences interleaved. Additional implementation is necessary to support such test cases.

Finally, the FARLEAD2 performance and effectiveness results under increasing test complexity may not generalize to overly complex test scenarios, owing to the explosion in the search space. If the shortest witness for a test scenario is too long for FARLEAD2 to find, the developer/tester might consider dividing the test scenario into two or more test scenarios.

VII. CONCLUSION

In summary, we have proposed FARLEAD2, a fast witness generation method for readable test scenarios using generalized experience replay (GER). We have described the novel staged test scenario (STS) language and explained how GER works with STS instances via flowcharts and examples. Our experiments have shown that FARLEAD2 generates a witness 95.7 times out of 100 and does it in 520 seconds, on average, indicating that FARLEAD2 is approximately 65 percent faster and 6.3 percent more effective than its best predecessor.

In the future, we will evaluate how different test scenario orderings affect witness generation and determine the best test scenario ordering for the developer/tester. We will execute GER, RL, and RND on large-scale AUTs to generalize our results further. We will train multiple RL agents with different test scenarios (hence, multi-objective) within the same execution, enabling simultaneous witness generation for many test scenarios. Finally, we will also conduct a manual testing study to evaluate the benefits of automated witness generation.

REFERENCES

[1] L. Ceci. (Jul. 13, 2022). *Average Number of New Android App Releases Via Google Play Per Month From March 2019 to June 2022*. [Online]. Available: https://www.statista.com/statistics/1020956/android-app-releases-worldwide/

[2] D. Bolton. (May 25, 2017). *88% of People Will Abandon an App Because of Bugs*. [Online]. Available: https://www.applause.com/blog/app-abanment-bug-testing

[3] O. Rodríguez-Valdés, T. E. J. Vos, P. Aho, and B. Marín, "30 years of automated GUI testing: A bibliometric analysis," in *Proc. QUATIC*, Algarve, Portugal, 2021, pp. 473–488.

[4] Google Developers. (Jan. 31, 2022). *Android UI/Application Exerciser Monkey*. [Online]. Available: https://developer.android.com/studio/test/tools/help/monkey.html

[5] S. Anand, M. Naik, M. J. Harrold, and H. Yang, "Automated concolic testing of smartphone apps," in *Proc. ACM SIGSOFT 20th Int. Symp. Found. Softw. Eng.*, Cary, NC, USA, 2012, pp. 1–11.

[6] T. Azim and I. Neamtiu, "Targeted and depth-first exploration for systematic testing of Android apps," in *Proc. ACM SIGPLAN Int. Conf. Object Oriented Program. Syst. Lang. Appl.*, Indianapolis, IN, USA, Oct. 2013, pp. 641–660.

[7] A. Machiry, R. Tahiliani, and M. Naik, "Dynodroid: An input generation system for Android apps," in *Proc. ACM FSE*, Saint Petersburg, Russia, 2013, pp. 224–234.

[8] W. Yang, M. R. Prasad, and T. Xie, "A grey-box approach for automated GUI-model generation of mobile applications," in *Proc. LNCS FASE*, Rome, Italy, 2013, pp. 250–265.

[9] W. Choi, G. Necula, and K. Sen, "Guided GUI testing of Android apps with minimal restart and approximate learning," in *Proc. ACM SIGPLAN Int. Conf. Object Oriented Program. Syst. Lang. Appl.*, Indianapolis, IN, USA, Oct. 2013, pp. 623–640.

[10] R. Mahmood, N. Mirzaei, and S. Malek, "EvoDroid: Segmented evolutionary testing of Android apps," in *Proc. 22nd ACM SIGSOFT Int. Symp. Found. Softw. Eng.*, Hong Kong, Nov. 2014, pp. 599–609.

[11] D. Amalfitano, A. R. Fasolino, P. Tramontana, B. D. Ta, and A. M. Memon, "MobiGUITAR: Automated model-based testing of mobile apps," *IEEE Softw.*, vol. 32, no. 5, pp. 53–59, Sep. 2015, doi: 10.1109/MS.2014.55.

[12] S. Hao, B. Liu, S. Nath, W. G. J. Halfond, and R. Govindan, "PUMA: Programmable UI-automation for large-scale dynamic analysis of mobile apps," in *Proc. 12th Annu. Int. Conf. Mobile Syst., Appl., Services*, Bretton Woods, NH, USA, Jun. 2014, pp. 204–217.

[13] M. Linares-Vasquez, M. White, C. Bernal-Cardenas, K. Moran, and D. Poshyvanyk, "Mining Android app usages for generating actionable GUI-based execution scenarios," in *Proc. IEEE/ACM 12th Work. Conf. Mining Softw. Repositories*, Florence, Italy, May 2015, pp. 111–122.

[14] K. Moran, M. Linares-Vasquez, C. Bernal-Cardenas, C. Vendome, and D. Poshyvanyk, "Automatically discovering, reporting and reproducing Android application crashes," in *Proc. IEEE Int. Conf. Softw. Test., Verification Validation (ICST)*, Chicago, IL, USA, Apr. 2016, pp. 33–44.

[15] K. Mao, M. Harman, and Y. Jia, "Sapienz: Multi-objective automated testing for Android applications," in *Proc. 25th Int. Symp. Softw. Test. Anal.*, Saarbrücken, Germany, Jul. 2016, pp. 94–105.

[16] N. Mirzaei, J. Garcia, H. Bagheri, A. Sadeghi, and S. Malek, "Reducing combinatorics in GUI testing of Android applications," in *Proc. 38th Int. Conf. Softw. Eng.*, May 2016, pp. 559–570.

[17] T. Su, G. Meng, Y. Chen, K. Wu, W. Yang, Y. Yao, G. Pu, Y. Liu, and Z. Su, "Guided, stochastic model-based GUI testing of Android apps," in *Proc. 11th Joint Meeting Found. Softw. Eng.*, Paderborn, Germany, Aug. 2017, pp. 245–256.

[18] Y. Cao, G. Wu, W. Chen, and J. Wei, "CrawlDroid: Effective model-based GUI testing of Android apps," in *Proc. 10th Asia–Pacific Symp. Internetware*, Beijing, China, Sep. 2018, pp. 1–6.

[19] Y. Koroglu, A. Sen, O. Muslu, Y. Mete, C. Ulker, T. Tanriverdi, and Y. Donmez, "QBE: QLearning-based exploration of Android applications," in *Proc. IEEE 11th Int. Conf. Softw. Test., Verification Validation (ICST)*, Västerås, Sweden, Apr. 2018, pp. 105–115.

[20] Y. Koroglu and A. Sen, "TCM: Test case mutation to improve crash detection in Android," in *Proc. LNCS FASE*, Thessaloniki, Greece, 2018, pp. 264–280.

[21] J. Yan, L. Pan, Y. Li, J. Yan, and J. Zhang, "LAND: A user-friendly and customizable test generation tool for Android apps," in *Proc. 27th ACM SIGSOFT Int. Symp. Softw. Test. Anal.*, Amsterdam, The Netherlands, Jul. 2018, pp. 360–363.

[22] C. Cao, J. Deng, P. Yu, Z. Duan, and X. Ma, "ParaAim: Testing Android applications parallel at activity granularity," in *Proc. IEEE 43rd Annu. Comput. Softw. Appl. Conf. (COMPSAC)*, Milwaukee, WI, USA, Jul. 2019, pp. 81–90.

[23] M. Pan, A. Huang, G. Wang, T. Zhang, and X. Li, "Reinforcement learning based curiosity-driven testing of Android applications," in *Proc. 29th ACM SIGSOFT Int. Symp. Softw. Test. Anal.*, Jul. 2020, pp. 153–164.

[24] H. N. Yasin, S. H. A. Hamid, and R. J. Raja Yusof, "DroidbotX: Test case generation tool for Android applications using Q-learning," *Symmetry*, vol. 13, no. 2, p. 310, Feb. 2021, doi: 10.3390/sym13020310.

[25] H. Guo, X. Liu, B. Li, L. Cai, Y. Hu, and J. Cao, "SQDroid: A semantic-driven testing for Android apps via Q-learning," in *Proc. IEEE 21st Int. Conf. Softw. Quality, Rel. Secur. (QRS)*, Hainan, China, Dec. 2021, pp. 301–310.

[26] E. Collins, A. Neto, A. Vincenzi, and J. Maldonado, "Deep reinforcement learning based Android application GUI testing," in *Proc. Brazilian Symp. Softw. Eng.*, Joinville, Brazil, Sep. 2021, pp. 186–194.

[27] M. K. Khan and R. Bryce, "Android GUI test generation with SARSA," in *Proc. IEEE 12th Annu. Comput. Commun. Workshop Conf. (CCWC)*, Las Vegas, NV, USA, Jan. 2022, pp. 487–493.

[28] M. M. Eler, J. M. Rojas, Y. Ge, and G. Fraser, "Automated accessibility testing of mobile apps," in *Proc. IEEE 11th Int. Conf. Softw. Test., Verification Validation (ICST)*, Västerås, Sweden, Apr. 2018, pp. 105–115.

[29] Y. Liu, C. Xu, S. C. Cheung, and J. Lu, "GreenDroid: Automated diagnosis of energy inefficiency for smartphone applications," *IEEE Trans. Softw. Eng.*, vol. 40, no. 9, pp. 911–940, Sep. 2014. Accesssed: Sep. 28, 2022, doi: 10.1109/TSE.2014.2323982.

[30] R. N. Zaeem, M. R. Prasad, and S. Khurshid, "Automated generation of oracles for testing user-interaction features of mobile apps," in *Proc. IEEE 7th Int. Conf. Softw. Test., Verification Validation*, Cleveland, OH, USA, Mar. 2014, pp. 183–192.

[31] T. Su, Y. Yan, J. Wang, J. Sun, Y. Xiong, G. Pu, K. Wang, and Z. Su, "Fully automated functional fuzzing of Android apps for detecting non-crashing logic bugs," in *Proc. ACM OOPSLA*, Chicago, IL, USA, 2021, pp. 1–31.

[32] Y. Li, Z. Yang, Y. Guo, and X. Chen, "DroidBot: A lightweight UI-guided test input generator for Android," in *Proc. IEEE/ACM 39th Int. Conf. Softw. Eng. Companion (ICSE-C)*, Buenos Aires, Argentina, May 2017, pp. 23–26.

[33] Y. Koroglu, A. Sen, and A. Akin, "Automated functional test generation practice for a large-scale Android application," in *Proc. Turkish Nat. Softw. Eng. Symp. (UYMS)*, Istanbul, Turkey, Oct. 2020, pp. 1–3.

[34] H. Mao, M. Alizadeh, I. Menache, and S. Kandula, "Resource management with deep reinforcement learning," in *Proc. 15th ACM Workshop Hot Topics Netw.*, Atlanta, GA, USA, Nov. 2016, pp. 50–56.

[35] I. Arel, C. Liu, T. Urbanik, and A. G. Kohls, "Reinforcement learning-based multi-agent system for network traffic signal control," *IET Intell. Transp. Syst.*, vol. 4, no. 2, pp. 128–135, 2010. Accessed: Sep. 28, 2022, doi: 10.1049/iet-its.2009.0070.

[36] D. Silver, T. Hubert, J. Schrittwieser, I. Antonoglou, M. Lai, A. Guez, M. Lanctot, L. Sifre, D. Kumaran, T. Graepel, T. Lillicrap, K. Simonyan, and D. Hassabis, "Mastering chess and Shogi by self-play with a general reinforcement learning algorithm," 2017, *arXiv:1712.01815*.

[37] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller, "Playing atari with deep reinforcement learning," 2013, *arXiv:1312.5602*.

[38] Z. Zhou, X. Li, and R. N. Zare, "Optimizing chemical reactions with deep reinforcement learning," *ACS Central Sci.*, vol. 3, no. 12, pp. 1337–1344, Dec. 2017, doi: 10.1021/acscentsci.7b00492.

[39] Y. Koroglu and A. Sen, "Reinforcement learning-driven test generation for Android GUI applications using formal specifications," 2019, *arXiv:1911.05403*.

[40] Y. Koroglu and A. Sen, "Functional test generation from UI test scenarios using reinforcement learning for Android applications," *Softw. Test., Verification Rel.*, vol. 31, no. 3, p. e1752, May 2021. Accessed: Sep. 28, 2022, doi: 10.1002/stvr.1752.

[41] I. Banerjee, B. Nguyen, V. Garousi, and A. Memon, "Graphical user interface (GUI) testing: Systematic mapping and repository," *Inf. Softw. Technol.*, vol. 55, no. 10, pp. 1679–1694, Oct. 2013. Accessed: Sep. 28, 2022, doi: 10.1016/j.infsof.2013.03.004.

[42] A. Memon, I. Banerjee, and A. Nagarajan, "What test Oracle should I use for effective GUI testing?" in *Proc. 18th IEEE Int. Conf. Automated Softw. Eng.*, Montreal, QC, Canada, Oct. 2003, pp. 164–173.

[43] P. S. N. Mindom, A. Nikanjam, and F. Khomh, "A comparison of reinforcement learning frameworks for software testing tasks," 2022, *arXiv:2208.12136*.

[44] A. M. Memon, "GUI testing: Pitfalls and process," *Computer*, vol. 35, no. 8, pp. 87–88, Aug. 2002. Accessed: Sep. 28, 2022, doi: 10.1109/MC.2002.1023795.

[45] A. M. Memon, M. E. Pollack, and M. L. Soffa, "A planning-based approach to GUI testing," in *Proc. QW*, San Fransisco, CA, USA, 2000, pp. 1–14.

[46] Y. Tsujino, "A verification method for some GUI dialogue properties," *Syst. Comput. Japan*, vol. 31, no. 14, pp. 38–46, Dec. 2000. Accessed: Sep. 28, 2022, doi: 10.1002/1520-684X(200012)31:14<38::AID-SCJ5>3.0.CO;2-R.

[47] F. Zaraket, W. Masri, M. Adam, D. Hammoud, R. Hamzeh, R. Farhat, E. Khamissi, and J. Noujaim, "GUICOP: Specification-based GUI testing," in *Proc. IEEE 5th Int. Conf. Softw. Test., Verification Validation*, Montreal, QC, Canada, Apr. 2012, pp. 747–751.

[48] D. Hammoud, F. A. Zaraket, and W. Masri, "GUICop: Approach and toolset for specification-based GUI testing," *Softw. Test., Verification Rel.*, vol. 27, no. 8, p. e1642, Dec. 2017. Accessed: Sep. 28, 2022, doi: 10.1002/stvr.1642.

[49] J. Wang. (Mar. 2022). *Detecting Non-Crashing Functional Bugs in Android Apps via Deep-State Differential Analysis*. [Online]. Available: https://tingsu.github.io/files/fse22-odin.pdf

[50] Z. Lv. (Mar. 2022). *Fastbot2: Reusable Automated Model-Based GUI Testing for Android Enhanced by Reinforcement Learning*. [Online]. Available: https://tingsu.github.io/files/ASE22-industry-Fastbot.pdf

[51] W. Liang and L. Li, "An automated planning approach to user interface model checking," in *Proc. Int. Conf. E-Product E-Services E-Entertainment*, Chennai, India, Nov. 2010, pp. 1–4.

[52] L. Mariani, M. Pezze, O. Riganelli, and M. Santoro, "AutoBlackTest: Automatic black-box testing of interactive applications," in *Proc. IEEE 5th Int. Conf. Softw. Test., Verification Validation*, Montreal, QC, Canada, Apr. 2012, pp. 81–90.

[53] S. Carino and J. H. Andrews, "Dynamically testing GUIs using ant colony optimization (T)," in *Proc. 30th IEEE/ACM Int. Conf. Automated Softw. Eng. (ASE)*, Lincoln, NE, USA, Nov. 2015, pp. 138–148.

[54] A. Esparcia-Alcázar, F. Almenar, M. Martínez, U. Rueda, and T. Vos, "Q-learning strategies for action selection in the TESTAR automated testing tool," in *Proc. META*, Marrakech, Morocco, 2016, pp. 130–137.

[55] Y. Falcone, Y. Falcone, S. Currea, and M. Jaber, "Runtime verification and enforcement for Android applications with RV-Droid," in *Proc. Springer RV*, Istanbul, Turkey, 2012, pp. 88–95.

[56] P. Daian, Y. Falcone, P. Meredith, T. F. Serbanuta, S. Shiriashi, A. Iwai, and G. Rosu, "RV-Android: Efficient parametric Android runtime verification, a brief tutorial," in *Proc. Int. Conf. Runtime Verification*, Vienna, Austria, 2015, pp. 342–357.

[57] H. Sun, A. Rosa, O. Javed, and W. Binder, "ADRENALIN-RV: Android runtime verification using load-time weaving," in *Proc. IEEE Int. Conf. Softw. Test., Verification Validation (ICST)*, Tokyo, Japan, Mar. 2017, pp. 532–539.

[58] P. Zhang, K. Cheng, and J. Gao, "Android-SRV: Scenario-based runtime verification of Android applications," *Int. J. Softw. Eng. Knowl. Eng.*, vol. 28, no. 2, pp. 239–257, Feb. 2018. Accessed: Sep. 28, 2022, doi: 10.1142/S0218194018500080.

[59] M. Hasanbeig, A. Abate, and D. Kroening, "Certified reinforcement learning with logic guidance," 2019, *arXiv:1902.00778*.

[60] M. Hasanbeig, A. Abate, and D. Kroening, "Logically-constrained neural fitted Q-iteration," in *Proc. AAMAS*, Montreal, QC, Canada, 2019, pp. 2012–2014.

[61] R. T. Icarte, T. Q. Klassen, R. Valenzano, and S. A. McIlraith, "Teaching multiple tasks to an RL agent using LTL," in *Proc. AAMAS*, Stockholm, Sweden, 2018, pp. 452–461.

[62] M. Wen, R. Ehlers, and U. Topcu, "Correct-by-synthesis reinforcement learning with temporal logic constraints," in *Proc. IEEE/RSJ Int. Conf. Intell. Robots Syst. (IROS)*, Hamburg, Germany, Sep. 2015, pp. 4983–4990.

[63] A. D. Laud, "Theory and application of reward shaping in reinforcement learning," Ph.D. dissertation, Comput. Sci. Dept., UIUC, Urbana-Champaign, IL, USA, 2004.

[64] L.-J. Lin, "Self-improving reactive agents based on reinforcement learning, planning and teaching," *Mach. Learn.*, vol. 8, nos. 3–4, pp. 293–321, May 1992. Accessed: Sep. 28, 2022, doi: 10.1007/BF00992699.
[65] T. Su, J. Wang, and Z. Su, "Benchmarking automated GUI testing for Android against real-world bugs," in *Proc. 29th ACM Joint Meeting Eur. Softw. Eng. Conf. Symp. Found. Softw. Eng.*, Athens, Greece, Aug. 2021, pp. 119–130.
[66] Y. Koroglu. (Aug. 11, 2019). *Black is Missing From the Color Palette (Issue #75)*. [Online]. Available: https://github.com/SecUSo/privacy-friendly-notes/issues/75

YAVUZ KOROGLU (Member, IEEE) received the B.S. and M.S. degrees in computer engineering from Boğaziçi University, Istanbul, Turkey, in 2014 and 2016, respectively.

From 2014 to 2018, he worked as a Teaching Assistant with the Department of Computer Engineering, Boğaziçi University. From 2018 to 2019, he was a Researcher with the Institute for Software Technologies, TU Graz, Austria. From 2019 to 2020, he worked as a Research Assistant with the Department of Computer Engineering, Boğaziçi University. Since 2020, he has been the Project Manager of Trailblu Inc., Turkey. He is the first author of several publications in top conferences and journals, including but not limited to IEEE International Conference on Software Testing, Verification and Validation (ICST'18), Fundamental Approaches to Software Engineering (FASE'18) Conference, and *Software Testing, Verification, & Reliability* (STVR) journal. His research interests include automated test generation, bug prediction, fuzzing, machine learning, reinforcement learning, and formal verification.

Mr. Koroglu was a recipient of two best paper awards, one from the National Software Engineering Symposium held in Turkey, in 2018, and the other from the 14th International Workshop on Automation of Software Test (AST) held in conjuction with the International Conference on Software Engineering (ICSE), in 2019.

ALPER SEN (Senior Member, IEEE) received the B.S. and M.S. degrees in electrical and electronics engineering from Middle East Technical University, Ankara, Turkey, in 1995 and 1997, respectively, and the Ph.D. degree in electrical and computer engineering from The University of Texas at Austin, Austin, TX, USA, in 2004.

He is currently a Full Professor with the Department of Computer Engineering, Boğaziçi University, Istanbul, Turkey. His current research interests include software testing, verification, and distributed systems.