2012 IEEE Fifth International Conference on Software Testing, Verification and Validation

AutoBlackTest: Automatic Black-Box Testing of Interactive Applications

Leonardo Mariani*, Mauro Pezzè*†, Oliviero Riganelli* and Mauro Santoro*

*University of Milano Bicocca
Viale Sarca, 336 - Milano, Italy

†University of Lugano
Via Buffi, 13 - Lugano, Switzerland

***Abstract*—Automatic test case generation is a key ingredient of an efficient and cost-effective software verification process. In this paper we focus on testing applications that interact with the users through a GUI, and present AutoBlackTest, a technique to automatically generate test cases at the system level. AutoBlackTest uses reinforcement learning, in particular Q-Learning, to learn how to interact with the application under test and stimulate its functionalities. The empirical results show that AutoBlackTest can execute a relevant portion of the code of the application under test, and can reveal previously unknown problems by working at the system level and interacting only through the GUI.**

***Keywords*-Black-Box Testing, Test Automation, Q-Learning**

I. Introduction

Automatically generating test cases can both reduce the costs and increase the effectiveness of testing. Current automatic test case generation techniques produce test cases that cover either functional or structural elements, reducing the human biases, and at a lower cost than manually generated and maintained test suites [1].

The problem of automatically generating test cases has attracted the attention of software engineers since the early seventies [2]. The difficulty and complexity of the problem has challenged researchers and practitioners for decades, and the problem is still largely open. Recent research has focused on several aspects of automatic test case generation that span from unit to regression testing, white to black-box testing, random to model-based testing, and cover different application domains, from embedded to telecommunication and interactive applications [3], [4], [5].

In this paper we consider interactive applications, that is applications that serve the requests of users who interact with the applications through either a Graphical User Interface (GUI) or the Web. These applications are widely used both in classic settings, like interactive desktop applications, and in the new framework of Web and mobile devices that offer a large number of such applications. In this paper, we focus on the problem of generating system test cases for applications that interact with the users through a GUI. For the sake of simplicity, we refer to such kind of applications as *applications*.

Current techniques for generating GUI test cases work in two phases. In the first phase, they generate a model of the event sequences that can be produced by interacting with the GUI of the application under test. In the second phase, they generate a set of test cases that cover the sequences in the model [6], [7]. Different techniques focus on different models and various coverage criteria, such covering system events [8] or semantically interacting events [9].

These techniques can automatically generate many test cases, whose effectiveness depends on the completeness of the initial model. When the initial model is obtained by stimulating the application under test with a simple sampling strategy that uses a subset of GUI actions to navigate and analyze the windows, the derived model is partial and incomplete, and the generated test cases necessarily overlook the many interactions and windows not discovered in the initial phase.

In this paper we present AutoBlackTest, a test case generation technique that builds the model and produces the test cases incrementally, while interacting with the application under test. AutoBlackTest discovers the most relevant functionalities and generates test cases that thoroughly sample these functionalities. It integrates two strategies, *learning* and *heuristics*. It uses Q-Learning [10] to turn the problem of generating test cases for an unknown application into the problem of an agent that must learn how to effectively act in an unknown environment. It uses multiple heuristics to effectively address a number of common but complex cases, such as compiling complex forms, executing the application with normal and abnormal values, and handling specific situations (for instance, saving files). AutoBlackTest aims to produce test cases that can activate relevant functional computations and that can likely reveal failures when executed. Since AutoBlackTest does not rely on an initial set of executions, it does not suffer from the limitations of current approaches.

The major contributions of this paper are:

- The definition of a technique for generating system test cases that can thoroughly sample the execution space of interactive GUI applications;
- The adaptation of Q-Learning to the problem of automatically generating test case for GUI applications;
- The empirical comparison of AutoBlackTest with GUITAR, that represents the current state of the art. The comparison is done on four applications for desktop computers. The results show that AutoBlackTest outperforms GUITAR both in terms of code coverage and number of revealed faults when used in overnight

978-0-7695-4670-4/12 $26.00 © 2012 IEEE
DOI 10.1109/ICST.2012.20

IEEE computer society

sessions.

The paper is organized as follow. Section II overviews the AutoBlackTest technique. Section III describes how Q-Learning is integrated in AutoBlackTest. Sections IV, V and VI describe in detail the Q-learning steps. Section VII describes the synthesis of the concrete test cases. Section VIII describes the failure detection mechanisms supported by AutoBlackTest. Section IX presents empirical results. Section X discusses related work. Section XI summarizes the contributions of the paper.

II. AutoBlackTest

AutoBlackTest is a technique and a tool for automatically generating systems test cases for interactive applications. The goal of AutoBlackTest is to generate test cases that exercise the functionalities of the application by navigating its GUI. The AutoBlackTest technique combines a Q-Learning Agent [10] with a Test Case Selector, as shown in Figure 1.

The *Q-Learning Agent* is responsible for interacting with the application under test. In particular, the agent aims to identify and pursue executions that result in relevant computations. The Q-Learning Agent works incrementally by selecting an action to execute on the target GUI, executing the action, observing the reaction of the application that consists of a new state of the GUI, and incorporating all these information in a model that is exploited to decide the next action to execute. The model represents the knowledge about the application that has been acquired by the agent, and it is incrementally extended during the testing process. Differently from state of the art approaches, AutoBlackTest learns the model incrementally by exploring the GUI and thus is not limited by the precision of an initial model.

The behavior of the Q-Learning Agent is disciplined by two main parameters, the number and the length of the episodes. The number of the episodes indicates the number of executions that the agent must produce. Test experts may set this value following different policies, for example according to the time available for the automatic generation of test cases. Since the Q-Learning Agent works incrementally and can be safely stopped at any time, the test experts can exploit at best the available time by leaving the tool running indefinitely, and manually stopping the agent when the time available for automatic test case generation expired. The length of the episodes is the number of actions that the agent should execute before stopping an episode and starting a new one.

In the current prototype, AutoBlackTest interacts with the GUI of the application under test by extending IBM Functional Tester [11], a commercial capture and reply tool. The current AutoBlackTest prototype can test the GUI frameworks supported by IBM Functional Tester, including JAVA, .NET and a number of other Windows and Linux GUI frameworks.

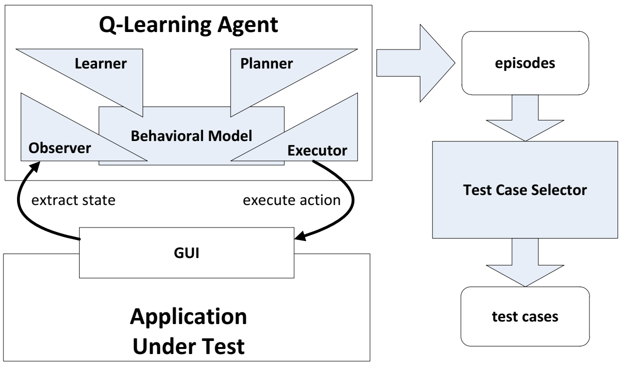

Figure 1. AutoBlackTest.

The episodes identified by the Q-Learning Agent correspond to test cases that can be used to validate future revisions of the application. The *Test Case Selector* is responsible for filtering the episodes executed by AutoBlackTest, removing the redundant ones, and generating a test suite that can be automatically re-executed with IBM Functional Tester.

III. Q-Learning Agent

The Q-Learning Agent is a standard implementation of Q-Learning [10] where we designed the Observer, the Learner, the Planner and the Executor to generate test cases for interactive applications. The architecture of the Q-Learning Agent matches the architecture of a classic autonomic component [12], as shown in Figure 1. The agent has failure detection capabilities that are not shown in Figure 1 and are discussed in Section VIII.

When executing a new episode, the Q-Learning Agent randomly selects a state of the model (the states of the model represent the GUI states of the application) and starts executing the episode from the selected state. The Q-Learning Agent brings the application into the selected state by executing a sequence of actions extracted from the model. These actions are not counted in the length of the executed episode. The behavior of the application along the executed sequence may be non-deterministic. If the sequence does not bring the application into the desired state, AutoBlackTest starts the episode from the reached state.

The possibility of starting the execution of an episode from any state is a standard assumption of Q-Learning algorithms, and is useful both to increase the likelihood of exploring the whole environment, in our case of the application under test, and reduce the chances to have the agent trapped in the initial states for most of its activity. Moreover, starting from a random state mitigates the issue of determining the length of the episodes because all the GUI states of the target application are likely reachable independently from the maximum length of the episodes.

The Q-Learning Agent iteratively determines the next action to execute in four steps. In *step 1*, the *Observer* analyzes the GUI of the application and extracts an abstract representation of the current state that is passed to the Learner. In a nutshell, the representation of the state is a collection of the property values of the GUI widgets of the applications. In *step 2*, the *Learner* updates the model according to the state reached by the execution, the action that has been executed, and the immediate utility of the action. We define the utility of actions in Section V. In *step 3*, the *Planner* uses the behavioral model to select the next action to execute. In *step 4*, the *Executor* executes the action selected by the Planner. The application reaches a new GUI state and a new iteration starts.

When specific conditions hold the Q-Learning process is known to converge to an optimal solution [13]. In our case, these conditions do not hold. This is not a problem because the agent can explore only a small portion of the entire execution space, and thus produces effective results in a limited time, regardless of convergence.

Section IV describes the Observer (which executes step 1). Section V describes the Learner (which executes step 2) and the behavioral model. Section VI describes the Planner and the Executor (which execute steps 3 and 4, respectively).

IV. Observer

The Observer builds an abstract representation of the state of the application. The abstract state is used as part of the model maintained by the Learner. Ideally, we would like to build a model that represents the entire state of the application. However, the whole state of an application is huge, since it includes the values of a number of variables, registries, persistent memories, etc., hard to access, and difficult to abstract. The Observer addresses this issue by approximating the state of the application with the portion of the state that is visible to the users, that is the GUI state.

In particular, a concrete GUI state S consists of a collection of widgets $\{w_1, \dots w_n\}$. Each widget w_i is a pair $(type_i, P_i)$, where $type_i$ is the type of the widget and P_i is a set of properties with their values. Each widget is usually associated with a large number of properties. Including all properties in the state information would produce huge state representations, thus the Observer builds an abstract state AS from a concrete state S using an abstraction function prj, $AS = prj(S)$. The function prj is a projection operator that is applied to every widget in the state S, $prj(S) = \{prj(w_1), \dots, prj(w_n)\}$. Given a widget w_i, prj extracts a subset of its properties, that is $prj(w_i) = (type'_i, P'_i)$, where $|P'_i| \leq |P_i|$ ($|\ |$ denotes the cardinality of a set).

For each type of widget, we defined the prj function to extract the properties that carry semantically useful information. For instance, for a `textarea`, we extract the values of the properties `text`, `enabled`, `editable` and `visible`. AutoBlackTest currently supports about 40 types of widgets, and covers the most standard elements that can be incorporated in a GUI. In the infrequent case of an unsupported widget, AutoBlackTest extracts a standard set of properties that includes `enabled`, `class` and `visible`. The abstract state returned by the abstraction process is the state representation that is incorporated in the behavioral model.

Beside extracting an abstract representation of the GUI state, the Observer checks if the same widgets occur in multiple GUI states, possibly with some modified properties. The Observer identifies occurrences of the same widget in multiple states by comparing the widget in a GUI with the widgets represented in the states of the behavioral model. To detect multiple occurrences of the same widget, we defined a function that generates traits from widgets. A trait is a subset of the widget properties that are expected to be both representative and invariant for the given widget. Using traits, we can formally define a concept of identity. Given two widgets w_1, w_2, we say that $w_1 =_t w_2$ iff $trait(w_1) = trait(w_2)$. For instance, the trait of a button includes the type of the widget (for instance, `Button`), the position of the button in the GUI hierarchy, and the label visualized on the button (for instance, "OK"). This strategy is similar to a feature offered by IBM Functional Tester to compare widgets [11].

Based on this definition we can define the following restriction operator between abstract states: Given $AS = \{w_1, \dots w_n\}$, $AS' = \{w'_1, \dots w'_{n'}\}$ abstract states, $AS \setminus_t AS' = \{w_i | w_i \in AS \wedge \nexists w_k \in AS' s.t.\ w_i =_t w_k\}$.

V. Learner and Behavioral Model

The Learner builds and updates the behavioral model according to the activity of the Agent. The behavioral model is a labeled multidigraph. A labeled multidigraph is a graph where pairs of nodes can be connected by multiple directed edges. In our models, the nodes of the graph represent the abstract GUI states of the application as returned by the Observer, the edges represent the transitions between states, and the labels associated with the edges represent both the actions that trigger the transitions and the estimated utility values of the action. The utility values represent the importance of the action for testing, as discussed below.

Transitions are triggered by the execution of the actions that label the corresponding edges. Every time the Agent observes a new state or executes a new action, it extends the model by adding the corresponding state or edge.

Each edge is weighted with an utility value, called *Q-value*, that represents the likelihood that the corresponding transition will generate complex interactions with the application, which can be extremely useful for system testing. The Q-value is assigned after the execution of the action that corresponds to the edge, and is computed according to both an immediate utility value and the Q-values of the actions associated with the edges exiting the state reached with the current edge, if the action leads to an existing state. The immediate utility value represents the immediate utility of the action, and is a real value computed by a *reward* function in the range $[0,1]$. In the following we first present the reward function that computes the immediate utility of an action, and then the computation of the Q-values.

The value of actions with respect to testing depends on the computations activated by the action. For example, once completed a form to add a new event in a personal agenda, the application usually displays the entire calendar, producing a major change in terms of displayed widgets, and enabling many new actions to be tested. On the contrary, actions like filling text areas or clicking on combo boxes cause small changes of the GUI state, and are thus less interesting for system testing. The reward function favors actions that activate relevant computations, and penalizes actions that activate marginal computations. To heuristically identify the actions that trigger the relevant computations, the reward function assigns high reward values to the actions that induce many changes in the abstract GUI state, and low reward values to the actions that induce few changes in the abstract GUI state.

The heuristic is not perfect, but Q-Learning handles well the exceptions. Starting each episode from a random state and executing many random actions reduce the chances that the agents waste time in uninteresting areas of the execution space, even if few useless actions are highly rewarded or vice versa.

To define the reward function, we need to introduce the diff_w function that computes the degree of change of a same widget when observed in two different states, and the diff_{AS} function that computes the degree of change between two abstract states.

Given two widgets $w_1 = (\mathit{type}, P_1)$ and $w_2 = (\mathit{type}, P_2)$, such that $w_1 =_t w_2$, the diff_w function computes the proportion of properties that changed their value:
$\mathit{diff}_w(w_1, w_2) = \frac{|P_1 \setminus P_2| + |P_2 \setminus P_1|}{|P_1| + |P_2|}$.

Given two abstract states AS_1 and AS_2 in this order, diff_{AS} computes the fraction of widgets that have changed from AS_1 and AS_2, taking into account both the widgets that occur only in the target state AS_2, and the ones that occur in both states with modified properties, but ignoring the widgets that disappear from the original state AS_1:

$$\mathit{diff}_{AS}(AS_1, AS_2) = \frac{|AS_2 \setminus_t AS_1| + \sum_{w_1 \in AS_1, w_2 \in AS_2, w_1 =_t w_2} \mathit{diff}_w(w_1, w_2)}{|AS_2|}$$

We now define the reward function. Given an abstract state AS_1 and an action a executed from AS_1, we denote with $\delta(AS_1, a)$ the state reached by executing a from AS_1. The reward of the action is the proportion of new widgets and widgets that changed their properties from the original to the target state: $\mathit{reward}(AS_1, a) = \mathit{diff}_{AS}(AS_1, \delta(AS_1, a))$. We do not consider the widgets that disappear when moving from the original to the target state to avoid too fast increments of the Q-Values of the actions that produce transitions among windows. When the increments of Q-values are too fast, the activity of the agents tend to focus too much on the actions that cause transitions between windows, ignoring the windows themselves, while moderate increases of the Q-values lead to better explorations of the state space.

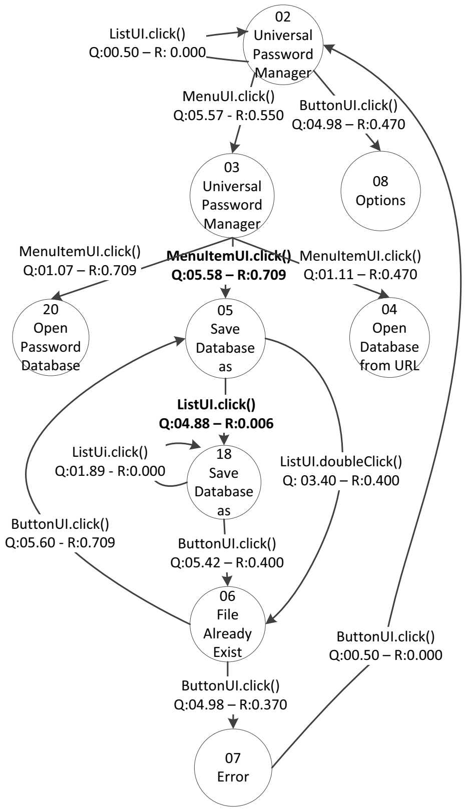

Figure 2. Excerpt of a model derived with AutoBlackTest

The reward function defined above can estimate well the immediate utility of an action, but does not take into account the value of the scenarios enabled by the execution of an action or a sequence of actions. For instance, a simple action that produces a transition between similar states, for example filling a text area, can enable the execution of actions that produce transitions between largely different states, for example successfully submitting a form. A testing process that aims to execute valuable computations needs to identify these transitions that potentially enable actions that induce large state changes later on. Q-learning captures these characteristics by annotating transitions with Q-values that represent the values of the actions computed by considering both the immediate utility as defined by the rewarding function, and the utility of the actions that can be executed in the future [10]. At each step, the Planner uses the Q-values to choose the action to be executed.

The role of Q-values is exemplified in the model shown in Figure 2. The model is a simplified excerpt of the model of the UPM application, an application for building a personal database of accounts, one of the case studies presented in this paper. The complete model is much larger and includes many more states and transitions, including additional transitions from the states represented in the figure. For the sake of simplicity in the figure, we label the states with the name of the UPM windows (instead of collection of widget properties), and the transitions with the type of the widget (instead of its identifier) followed by the name of the action (omitting parameters).

The path through states 02, 03, 05, 18, 06 represents the case of a user who overwrites an existing database of passwords with a new one, by selecting the `Save New...` command from a menu, clicking on the last existing database in a `FileChooser` window, clicking the `Save` button and finally confirming the intention to overwrite the existing database. The path reaches an error state because it corresponds to the attempt to overwrite the currently opened database. The edges are labeled with the actions that trigger the transition between states and the Q-values of the corresponding actions. For the sake of the presentation in this figure, we annotate the transitions also with the reward value of the action. The transitions with bold labels indicate that high Q-values can be assigned both to actions with high reward (for instance, the selection of the `Save New...` command, which enables an interesting scenario) and actions with low reward (for instance, selecting an existing database which does not produce relevant computations, but enables the overwriting of an existing database).

Q-Learning can also effectively identify alternative paths. For instance the action `ListUI.doubleClickFirst()` has a high Q-value because it combines the selection of an existing file and the click of the `Save` button into a single action, and terminates with the same state obtained by executing the entire sequence.

Q-values are computed by the Learner starting from the reward values according to standard Q-Learning formula:

$$Q(s,a) = \mathit{reward}(s,a) + \gamma \max_{a'} Q(\delta(s,a),a') \qquad (1)$$

where $\delta(s,a)$ is the state reached by executing the action a from the state s, $Q(\delta(s,a),a')$ indicates the Q-value associated with the action a' when executed from the state $\delta(s,a)$, $\mathit{reward}(s,a)$ indicates the immediate reward of the action a executed from the state s, and γ is a parameter called discount factor. The parameter γ is a real value in the range $[0,1]$ and is used to balance the relevance of the immediate reward with respect to future actions. In particular, a value close to 1 gives high weight to future actions, while a value close to 0 gives high weight to immediate rewards. In Q-Learning the parameter γ is usually closer to 1 than 0, because an agent aims to execute sequences of actions that maximize the reward collected during an entire episode, rather than maximizing the immediate reward. According to our experience with AutoBlackTest a good choice of the parameter is $\gamma = 0.9$. Other studies in Reinforcement Learning [14], [15] also confirm that a value of $\gamma = 0.9$ produces highly effective results.

When the action reaches a new state, the second term of the formula is zero and the Q-value is given by the reward function. Otherwise, the Q-value is computed according to both the reward function and the Q-values of the edges exiting the target state.

The Learner applies the formula to update the Q-Value every time an action is executed.

VI. Planner and Executor

The *Planner* selects the action executed at each iteration following the popular *ϵ-greedy* Q-Learning policy [14], [15].

The ϵ-greedy policy consists of selecting either a random action among the ones executable in the current GUI, with probability ϵ, or the best action that is the action with the highest Q-value according to the behavioral model available to the agent, with probability $1-\epsilon$. The value chosen for ϵ can impact significantly the effectiveness of the technique. We determined a suitable value for ϵ by empirically comparing different configurations for the ϵ-greedy policy when used in AutoBlackTest. The empirical study reported in Section IX shows that the 0.8-greedy policy produces the best results.

When the *Planner* selects a random action, it uses different probabilities for the available actions depending on their complexity. The *Planner* distinguishes between simple and complex actions. A *simple action* is a single event triggered on a single widget. A *complex action* is a workflow of simple actions orchestrated heuristically. Complex actions are defined to effectively face *specific situations*. For instance, interacting with an action that saves a window is more effective if a proper complex action has been designed, rather than only using simple actions that might frequently produce invalid file names. A complex action has an *enabling condition* that specifies under which condition the complex action can be executed and that consists of a number of constraints on the number, type and properties of the widgets in the current windows. If all these constraints evaluate to True, the complex action can be selected by the Planner. When both simple and complex actions are executable in the current state, the *Planner* select selects a complex one with a probability of 75%.

The *Executor* concretely interacts with the widgets and is activated to execute either a simple or a complex action. In the current prototype, the Executor interacts with the GUI using IBM Functional Tester.

In the following we describe the simple and complex actions currently supported by AutoBlackTest.

Simple Actions: Table I summarizes the set of supported widgets and the actions that AutoBlackTest can execute on these widgets. The 18 widgets reported in the table are macro classes of widgets that cover about 40 different specific widgets. Some of the actions require a

widget	action
Label, ToolTip, Button	click()
ToggleButton, Checkbox, RadioButton	select(), deselect()
TextField, FormattedText, TextArea, TextPane, EditorPane, PasswordField	write(text)

ComboBox, Tree, List, Table	click(pos), doubleClick(pos)
	click(elem), doubleClick(elem)

TabbedPane, Menu	click(elem)

Table I
SIMPLE ACTIONS.

parameter to be executed. For instance, a `Combobox` requires the specification of the element tot be clicked, which is identified either by specifying an element or its position. AutoBlackTest handles parameter values by using values from pre-defined sets of literals. The sets of literals have been defined following the boundary testing principle, that is a combination of exceptional, boundary and normal values. For example, the `Combobox` can be executed by clicking on the first, middle, or the last item in the widget (using `click(pos)`). In the future we aim at employing more sophisticated strategies than using pre-defined values, for instance exploiting GUI analysis algorithms [16].

Complex Actions: Table II specifies the complex actions currently supported by AutoBlackTest. For each action, the table reports the name of the action, the situation faced by the action, the conditions that enable the action and the behavior of the action.

VII. TEST CASE SELECTOR

The episodes automatically executed by AutoBlackTest during the testing process are test cases. These test cases can be stored in an executable form, for instance to support regression testing. Since AutoBlackTest records every information necessary to replicate the execution of an episode, the generation of executable IBM Functional Tester test cases is straightforward.

Episodes that execute the same sequences of actions are redundant. To produce a non-redundant test suite, the Test Case Selector filters episodes before generating the actual test cases. To this end, we select the episodes to be included in the test suite following the *additional statement coverage prioritization* approach [17]: AutoBlackTest selects the episode with the highest coverage, adds it to the test suite, recompute the marginal coverage of the remaining episodes and proceed by selecting a new episode until no episodes further contribute to code coverage. Episodes that do not contribute to code coverage according to this process are classified as redundant and discarded.

VIII. FAILURE DETECTION STRATEGIES

AutoBlackTest can detect both domain independent failures, like crashes, hangs and uncaught exceptions, and failures that cause violations of assertions, if available in the code. When detecting a failure, AutoBlackTest automatically produces a report. Here we describe the detection strategy for domain independent failures.

Crashes: AutoBlackTest detects a system crash by recognizing that the target application is not available anymore. It reports the sequence of actions that leads to the failure so that the failure (if deterministic) can be reproduced, interrupts the current episode, and starts a new one (starting a new episode implies restarting the application).

Hangs: AutoBlackTest detects hangs by recognizing that the target application is available, but is not responding to actions. It reports the sequence of actions that leads to the failure so that the failure (if deterministic) can be reproduced, interrupts the current episode, and starts a new one (restarting the application).

Uncaught Exceptions: AutoBlackTest monitors the standard output and error streams looking for exceptions. If it reveals an exception, it produces a failure report that allows the replication of the execution that exposed the failure (if deterministic). Differently from the previous cases, it does not interrupt the execution of the episode.

IX. EMPIRICAL EVALUATION

We evaluated AutoBlackTest by first studying the effectiveness of the ϵ-greedy policy adopted in AutoBlackTest for Q-Learning and discussed in Section VI, and then investigating the effectiveness of AutoBlackTest in supporting system testing. We estimated the capability of supporting system testing by measuring the code coverage achieved with AutoBlackTest and by evaluating the ability of finding uncovered problems in the target application. We compare the results with GUITAR that represents the state of the art in the field [8]. In the empirical evaluation we addressed the following research questions: (RQ1) What is a proper configuration for the ϵ-greedy policy used in AutoBlackTest? (RQ2) Can AutoBlackTest achieve a better code coverage than GUITAR? (RQ3) Can AutoBlackTest detect more faults than GUITAR?

We answered these questions by experimenting with four applications for desktop machines. We choose applications of different domains already exploited in similar studies[1]: UPM v1.6[2], a personal password manager (2.515 loc); PDFSAM v0.7 stable release 1[3], a tool for merging and splitting PDF documents (3.138 loc); TimeSlotTracker v0.4[4], an advanced manager of personal tasks and activities (3.499 loc); and Buddi v.3.4.0.8[5], a personal finance and budgeting program (10.580 loc).

We investigated the ϵ-greedy policy (RQ1) with UPM because it is the simplest of the selected applications, thus

[1] see for instance the GUITAR web page http://sourceforge.net/apps/mediawiki/guitar/

[2] http://upm.sourceforge.net/

[3] http://sourceforge.net/projects/pdfsam/

[4] http://sourceforge.net/projects/timeslottracker/

[5] http://buddi.digitalcave.ca/

action name	situation	enabling condition	behavior
File Chooser	This action is designed to interact with a open/save window (which is a standard modal window provided by most GUI frameworks). The purpose is to limit the scope of the interaction with the file system and produce legal file names.	The open/save window is detected by checking if the name of the class that implements the window matches one of the known open/save windows classes.	This action executes a random sequence of simple actions with the following constraints: it does not allow to move up in the folder hierarchy and a predefined set of valid filenames can be entered.
Color Chooser	This action is designed to interact with a color choosing window (which is a standard modal window provided by most GUI frameworks). The purpose is to support the particular grid that is displayed in this kind of window, and that it would not be usable with simple actions.	The color choosing window is detected by checking if the name of the class that implements the window matches one of the known color choosing classes.	This action can click either the color in the center of the grid or a color at the side of the grid. After the first click, this complex action clicks the ok button. If, instead of returning in the underlying window, a new window is opened, the action assumes that the window is an error window and closes the new window first, and finally clicks the cancel button in the color chooser window.
Fill Form	This action is designed to interact with form-like windows. The rational for this action is that form-like windows often require filling many/all the widgets with some data before clicking an ok button, to activate the corresponding functionality. Relying on a lucky combination of simple actions to obtain a sequence that fills most widgets before clicking a button is unrealistic.	A form-like window is identified by checking if at least 8 widgets that allow entering data (these widgets do not only consist of `textArea`, but also `comboBoxes`, `Trees`, etc.) and a button are active in the current window.	This action can have 6 behaviors obtained by combining the value of two parameters. The first parameter represents the percentage of the widgets that allow entering data that will be filled, and it can be assigned with 50%, 75% and 100%. The second parameter is a Boolean value that indicates whether the complex action should end by clicking the ok button, or should continue with the regular execution, without clicking the ok button.
HandleList	This action is designed to interact with a listbox in a way richer than executing a simple action. This is especially needed to select multiple items in listboxes.	This action is enabled if a listbox that allows the selection of multiple items is active in the current window.	This action can have three behaviors. It can select two items, it can select half of the items in the list or all the items in the list.
Compund<*>	This action represents a family of complex actions. We use the symbol <*> to indicate any kind of widget that allows to input data. This action is used to interact with windows that display multiple widgets of the same kind. The hypothesis is that it is likely necessary to fill many of them to activate an interesting computation. For instance, if a window displays several `comboBoxes`, it is likely necessary to make a choice for most of them to run the underlying function.	If at least 3 widgets of the same kind are active in the current window the action is activated.	This action can have three behaviors. It can fill 50%, 75% or 100% of the widgets.

Table II
COMPLEX ACTIONS.

strategy	$\epsilon = 0.6$	$\epsilon = 0.7$	$\epsilon = 0.8$	$\epsilon = 0.9$	$\epsilon = 1$
ϵ-greedy	84%	79%	**87%**	84%	81%

Table III
STATEMENT COVERAGE ACHIEVED WITH DIFFERENT POLICIES.

we expect the results be influenced more by the policy than by the actions. We studied code coverage (RQ2) and failure revealing ability (RQ3) with all four applications, and we compared the results with GUITAR.

RQ1: Action Selection Strategy

Since executing a significant number of random actions is important, we investigated the effectiveness of the ϵ-greedy policy for high values of ϵ. In particular, we considered the values: 0.6, 0.7, 0.8, 0.9 and 1.

We compared the effectiveness of the configurations referring to statement coverage that measures the amount of code explored by the test cases generated with AutoBlackTest. We collected the empirical data by executing AutoBlackTest three times for each configuration, limiting the length of each execution to 12 hours and using episodes of length 31.

The results are summarized in Table III, and indicate that all configurations achieve good statement coverage with a maximum for $\epsilon = 0.8$ that we chose as reference value for the experiments.

RQ2: Code Coverage

AutoBlackTest and GUITAR provide different results depending on the execution interval. In our empirical study, we assumed the two techniques to be used in overnight sessions, and compared the results produced after 12 hours of execution on an Intel i5 760@2.80 Ghz with 4GB ram.

Since AutoBlackTest generates test cases incrementally, we simply interrupted its execution after 12 hours of execution time. We executed AutoBlackTest with episodes of length 31 and with a 0.8-greedy policy.

GUITAR can generate multiple models. We setup GUITAR to use the Event Interaction Graph (EIG) as model for test case generation, since it increases the fault detection effectiveness of the generated test case, according to the authors [7], [18]. We ran a few testing sessions with different initial models to confirm that EIG is the best option. GUITAR generates abstract test cases that are later turned into concrete test cases. The generation process is influenced by the length of the abstract test cases, which are shorter than the corresponding concrete test cases. The authors of the GUITAR paper suggest to use abstract test cases no longer than 10 [7]. To find an appropriate length for the experiment, we applied GUITAR to the case studies looking for configurations that terminates the test case generation in at most 6 hours (50% of the budget). We discovered that with a value of 5 we can generate test cases within the time limit

App.	Technique	Stm. Cov.	♯ Test Cases
UPM	AutoBlackTest	86%	27
	GUITAR	73%	3456
PDFSAM	AutoBlackTest	64%	28
	GUITAR	53%	4500
TimeSlot Tracker	AutoBlackTest	68%	14
	GUITAR	55%	5140
Buddi	AutoBlackTest	59%	52
	GUITAR	41%	850

Table IV
COVERAGE DATA.

for every application. We use the remaining time, which has never been less than 6 hours with an average of 10 hours, to execute the tests. GUITAR was not able to interact with some frequently used windows in UPM (in particular the login window) and Buddi (in particular the donate window) and most of the executions got stuck soon. Using GUITAR as it is would have produced poor score for these two case studies. To compare the two techniques, we slightly modified the applications to let GUITAR proceeds when facing these two windows and avoid getting stuck.

Table IV shows the amount of statement covered in the experiments (column Stm. Cov.), and the size of the test suite automatically generated with AutoBlackTest and GUITAR (column ♯ test cases). We computed coverage after eliminating the lines of code that are trivially unreachable, like the methods that are never invoked by the application.

AutoBlackTest covered from 59% (PDFSAM) to 86% (UPM) of the statements, with an average of 69%. These results show that AutoBlackTest samples well the behavior of the applications. AutoBlackTest outperformed GUITAR in every case study: GUITAR covered from 41% to 73% of the statements, with an average of 55%. The code areas that GUITAR does not cover result from the intrinsic limitation of using an initial model to generate test cases, as done in GUITAR, but not in AutoBlackTest that builds its model by heuristically learning the shape of the execution space.

Figure 3 shows how AutoBlackTest incrementally increases statement coverage. As expected the amount of covered statements increases fast with the first episodes, up to about episode 70, and improves slowly afterwards (in each case study, the last 100 episodes increase coverage by about 5%). These results suggest that AutoBlackTest could be extended with an adaptive early termination policy, in the presence of strong time constraints. We can for example terminate AutoBlackTest when the coverage has not increased more than a given threshold (for instance 2%) in the last N episodes (for instance 50).

The AutoBlackTest test case selector successfully pruned the many executed episodes (about 180 in every case study) by eliminating redundant test cases and producing compact test suites that can be re-executed with IBM Functional Tester: For three out of the four applications the test case selector distilled less than 30 test cases that cover the same statements covered by the 180 episodes (17% of the episodes), and less than 60 test cases for the fourth application (33% of the episodes). GUITAR covers less statements generating many more test cases, 850 in the best case and 5140 in the worst case. We can conclude that AutoBlackTest is more effective than GUITAR in synthesizing a smaller set of test cases with high code coverage (180 before selection, and from 14 to 52 after test case selection).

App.	Exec	AutoBlackTest Faults			GUITAR Faults		
		sev	min	tot	sev	min	tot
UPM	ex 1	2	3	5	0	1	1
	ex 2	4	4	8	0	1	1
	ex 3	3	3	6	0	1	1
	avg (tot)			**6.3 (8)**			**1 (1)**
PDFSAM	ex 1	0	1	1	0	0	0
	ex 2	0	1	1	0	0	0
	ex 3	0	1	1	0	0	0
	avg (tot)			**1 (1)**			**0 (0)**
TimeSlot Tracker	ex 1	1	3	4	0	1	1
	ex 2	1	3	4	0	2	2
	ex 3	1	3	4	0	3	3
	avg (tot)			**4 (5)**			**2 (4)**
Buddi	ex 1	2	2	4	0	0	0
	ex 2	1	3	4	0	0	0
	ex 3	1	2	3	0	0	0
	avg (tot)			**3.6 (6)**			**0 (0)**

Table V
AMOUNT OF FAULTS REVEALED BY EXECUTING AUTOBLACKTEST AND GUITAR

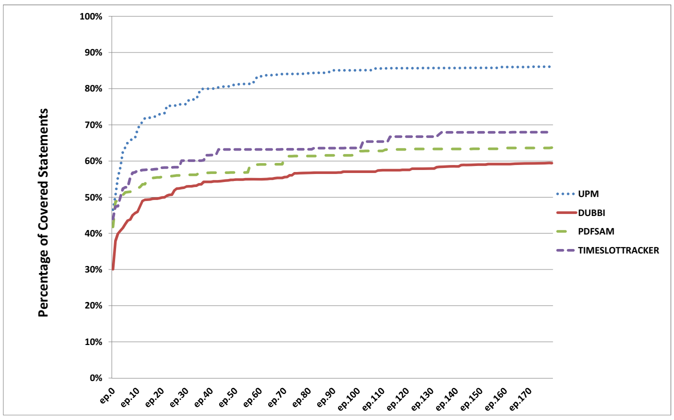

Figure 3. Average increment of covered statements along episodes.

RQ3: Fault Detection

We complete the evaluation of AutoBlackTest and GUITAR by examining the number of problems that AutoBlackTest and GUITAR detected automatically while running with the configuration described in RQ2. Table V shows the amount of faults detected in each execution (column tot) distinguishing between severe (column sev) and minor faults (column min). We classify faults as severe if they prevent the execution of a functionality, and as minor otherwise. We also report the average number of faults detected per execution (row avg) and the total number of faults detected across the three executions (row (tot)).

AutoBlackTest detected faults in all applications: from 1 (PDFSAM) to 8 (UPM) faults. The faults detected with AutoBlackTest were all present in the released applications.

14 out of the 20 detected faults are new faults not reported in the bug repositories of the analyzed applications. GUITAR detected faults only in 2 applications: 1 fault in UPM and 4 in TimeSlotTracker. From our empirical validation we can conclude that AutoBlackTest is more effective than GUITAR in detecting faults when used in a 12 hours time frame: AutoBlackTest detected a total of 20 faults while GUITAR detected a total of 5 faults (4 out of the 5 faults were also detected by AutoBlackTest).

The results presented in this section indicate that AutoBlackTest can be useful both to automatically test interactive applications and to support testers in detecting system level faults overlooked by alternative verification activities. We found particularly interesting the ability of AutoBlackTest to detect faults that can be detected only through rare combinations of actions, as frequently happens for system level faults. For instance, one of the faults detected in UPM leads to a failure only when a user first creates two accounts with an empty account name and then deletes the second account. This sequence of actions results in the impossibility of modifying the first account. Another interesting example is the fault detected in PDFSAM where a specific combination of inputs in a form-like window produces no response from the application. The lack of reaction from the application leaves the user with the impossibility to know if a specified PDF file has been split or not.

A. Threats to Validity

An important threat to the internal validity of the experiments is the selection of the case studies. To mitigate the risk of choosing applications that may impact the results, we selected third-party publicly available applications that have been already used in empirical studies about GUI testing, that have various sizes and cover different domains.

Another threat to internal validity is related to the definition of the parameters that influence the behavior of AutoBlackTest. To reduce this threat we chose the parameter values according to the empirical experiences reported in previous research (the discount factor), and the empirical analyses reported in this paper (the action selection). Both previous research and the experiments reported in this paper show that the performance of Q-Learning changes gracefully for small changes of these parameters, and thus the impact of a choice on the results is limited. We expect the results to be stable with respect to small changes of the configurations.

A final threat to internal validity is the setup of the GUITAR tool. To mitigate this risk, we configured the tool according to suggestions reported by authors of the GUITAR papers. We also executed quick test sessions to empirically confirm the validity of the suggestions in the context of our studies.

The main threat to the external validity comes from the limited number of case studies that we considered. We experimented AutoBlackTest with four applications, and used one of them to investigate the policy for action selection. Further studies are necessary to generalize the results, but the consistency of the results obtained so far with applications from different domains and of different size gives us good confidence about the effectiveness of AutoBlackTest.

The higher effectiveness of AutoBlackTest with respect to GUITAR has been experienced only simulating overnight test sessions of 12 hours. These results cannot be generalized to sessions of different length. In particular, they cannot be generalized to longer sessions and further studies are needed in this direction.

Finally, the main threat to the construction validity is related to the possible presence of faults in our implementation of AutoBlackTest [19]. To mitigate this threat we both manually checked the correctness of the behavioral model extracted for a number of sample executions and we validated the detected faults by replaying all the executions that detected issues in the case studies.

X. Related Work

Generating and executing system test cases for interacting applications is commonly a manual activity. Automation is mostly limited to capture and replay tools that re-execute previously recorded test cases. There are many capture and reply tools that work for several GUI frameworks, such as IBM Functional Tester [11] and Abbot [20]. These tools reduce regression testing effort, but rely on the manual generation and execution of the initial test suite. AutoBlackTest complements these tools by automatically generating and executing test cases for interactive applications.

Recent work on automated test generation exploits GUI coverage metrics that measure how many of the events produced by widgets have been stimulated in the testing process [6], [7]. These metrics evaluate how many widgets a test suites “touched”, but do not provide information about the computations that have been covered. For this reason, we used standard statement coverage to measure the amount of code executed by AutoBlackTest.

Memon et al. take advantage from GUI coverage metrics to define techniques that generate system test cases that cover sets of GUI events [8], [9]. The effectiveness of these techniques is limited by the accuracy of the initial model. AutoBlackTest overcomes this issue by building a model dynamically and incrementally while exploring the behavior of the application.

Xie and Memon investigated the use of oracles for GUI test cases. The empirical evaluation reported in [21] shows that there exist a tradeoff between the accuracy of the oracles and the cost of evaluating oracles at run-time. Several of the oracles evaluated by Xie and Memon can also be potentially integrated in AutoBlackTest.

Test case generation can benefit from additional information about the application under test. For instance, generation of system test cases can take advantage of usage profiles to be more effective [22]. Unfortunately good usage profiles are not always available.

Generation of system test cases has been investigated also in other domains. In particular, there are several techniques to generate test cases for Web applications. Some of these techniques share with GUI testing the underlying idea of covering specific sequences of events, for instance semantically interacting events [23]. Other techniques produce test cases by relying on navigation models [24] or data captured from users sessions [25]. While these models and data are quite common for Web applications, they are less frequently available for GUI applications.

XI. Conclusions

This paper presents AutoBlackTest, a technique for the automatic generation of system test cases for interactive applications. The results presented in this paper show that AutoBlackTest can automatically generate system test cases that cover a relevant portion of the statements in the target applications and discover faults not detected by developers, when applied to personal applications. We empirically compared AutoBlackTest with GUITAR when used in 12 hours test sessions, and we discovered that AutoBlackTest can generate test cases that cover more statements and reveal more failures than the test cases generated with GUITAR.

References

[1] M. Pezzè and M. Young, *Software Testing and Analysis: Process, Principles and Techniques.* John Wiley, 2007.

[2] J. King, "Symbolic execution and program testing," *Communications of the ACM*, vol. 19, no. 7, pp. 385–394, 1976.

[3] N. Tillmann and J. D. Halleux, "Pex: white box test generation for .NET," in *proceedings of the 2nd International Conference on Tests and Proofs.* Springer-Verlag, 2008.

[4] M. Broy, B. Jonsson, J.-P. Katoen, M. Leucker, and A. Pretschner, Eds., *Model-based Testing of Reactive Systems*, ser. LNCS, vol. 3472. Springer Verlag, 2005.

[5] K. Taneja and T. Xie, "DiffGen: Automated regression unit-test generation," in *proceedings of the 23rd International Conference on Automated Software Engineering*, 2008.

[6] A. M. Memon, M. L. Soffa, and M. E. Pollack, "Coverage criteria for GUI testing," in *proceedings of the 8th European Software Engineering Conference held jointly with 9th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2001.

[7] X. Yuan, M. Cohen, and A. M. Memon, "GUI interaction testing: Incorporating event context," *IEEE Transactions on Software Engineering*, vol. 37, no. 4, pp. 559–574, 2011.

[8] A. M. Memon and Q. Xie, "Studying the fault-detection effectiveness of GUI test cases for rapidly evolving software," *IEEE Transactions on Software Engineering*, vol. 31, no. 10, pp. 884–896, 2005.

[9] X. Yuan and A. M. Memon, "Generating event sequence-based test cases using GUI run-time state feedback," *IEEE Transactions on Software Engineering*, vol. 36, no. 1, pp. 81–95, 2010.

[10] R. Sutton and A. Barto, *Reinforcement Learning: An Introduction.* MIT Press, 1998.

[11] IBM, "IBM rational functional tester," http://www-01.ibm.com/software/awdtools/tester/functional/.

[12] J. O. Kephart and D. M. Chess, "The vision of autonomic computing," *IEEE Computer*, vol. 43, no. 1, pp. 41–50, 2003.

[13] C. J. C. H. Watkins and P. Dayan, "Technical note Q-Learning," *Machine Learning*, vol. 8, pp. 279–292, 1992.

[14] O. Abul, F. Polat, and R. Alhajj, "Multiagent reinforcement learning using function approximation," *IEEE Transactions on Systems, Man, and Cybernetics, Part C: Applications and Reviews*, vol. 30, no. 4, pp. 485 –497, 2000.

[15] L.-J. Lin, "Reinforcement learning for robots using neural networks," Ph.D. dissertation, Carnegie Mellon University, Pittsburgh, PA, USA, 1992.

[16] G. Becce, L. Mariani, O. Riganelli, and M. Santoro, "Extracting widget descriptions from GUIs," in *proceedings of the International Conference on Fundamental Approaches to Software Engineering*, 2012.

[17] G. Rothermel, R. H. Untch, C. Chu, and M. J. Harrold, "Prioritizing test cases for regression testing," *IEEE Transactions on Software Engineering*, vol. 27, no. 10, pp. 929–948, 2001.

[18] Q. Xie, "Developing cost-effective model-based techniques for gui testing," PhD Thesis, University of Maryland, 2006.

[19] L. Mariani, M. Pezzè, O. Riganelli, and M. Santoro, "AutoBlackTest: A tool for automatic black-box testing," in *proceedings of the International Conference on Software Engineering - Tool Demo*, 2011.

[20] T. Wall, "Abbot Java GUI test framework," http://abbot.sourceforge.net/.

[21] Q. Xie and A. M. Memon, "Designing and comparing automated test oracles for GUI-based software applications," *IEEE Transactions on Software Engineering*, vol. 16, no. 1, pp. 1–36, 2007.

[22] A. Brooks and A. Memon, "Automated GUI testing guided by usage profiles," in *proceedings of the International Conference on Automated Software Engineering*, 2007.

[23] A. Marchetto, P. Tonella, and F. Ricca, "State-based testing of Ajax web applications," in *proceedings of the International Conference on Software Testing, Verification, and Validation*, 2008.

[24] A. Andrews, J. Offutt, and R. Alexander, "Testing Web applications by modeling with FSMs," *Software and System Modeling*, vol. 4, no. 3, 2005.

[25] S. Elbaum, S. Karre, and G. Rothermel, "Improving Web application testing with user session data," in *proceedings of the International Conference on Software Engineering*, 2003.