SOFTWARE TESTING, VERIFICATION AND RELIABILITY
Softw. Test. Verif. Reliab. (2014)
Published online in Wiley Online Library (wileyonlinelibrary.com). DOI: 10.1002/stvr.1538

Automatic testing of GUI-based applications

Leonardo Mariani[1,*,†], Mauro Pezzè[1,2], Oliviero Riganelli[1] and Mauro Santoro[1]

[1] *University of Milano Bicocca, viale Sarca 336, Milano, Italy*
[2] *University of Lugano, via Buffi 13, Lugano, Switzerland*

SUMMARY

Testing GUI-based applications is hard and time consuming because it requires exploring a potentially huge execution space by interacting with the graphical interface of the applications. Manual testing can cover only a small subset of the functionality provided by applications with complex interfaces, and thus, automatic techniques are necessary to extensively validate GUI-based systems. This paper presents 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. When used to complement the activity of test designers, AutoBlackTest reuses the information in the available test suites to increase its effectiveness. The empirical results show that AutoBlackTest can sample better than state of the art techniques the behaviour of the application under test and can reveal previously unknown problems by working at the system level and interacting only through the graphical user interface. Copyright © 2014 John Wiley & Sons, Ltd.

Received 13 November 2012; Revised 11 March 2014; Accepted 25 April 2014

KEY WORDS: black-box testing; test automation; Q-learning

1. INTRODUCTION

GUI testing consists of testing applications through their graphical user interface (GUI). GUI testing is usually a manual activity and often concerns with the execution of system and acceptance test cases. Automatic test case generation has been largely investigated for many tasks, including unit, regression and model-based testing [1–3], but less thoroughly for GUI testing yet.

This paper considers interactive applications, that is applications that serve the requests of users who interact with the applications through a GUI. State of the art testing approaches of interactive applications are either black-box or white-box. Black-box techniques work in two steps: they first generate a model of the event sequences that can be produced by interacting with the GUI of the application under test and then generate a set of test cases that cover the sequences in the model [4, 5]. Different techniques refer to different models and coverage criteria, such as covering system events [6] or semantically interacting events [7].

White-box techniques use the information in the source code to generate test cases that cover code elements. System testing must traverse many software layers, and white box techniques exploit some heuristics to tackle the complexity of this task. Recent approaches in this category use either search-based [8] or concolic techniques [9].

The effectiveness of black-box techniques depends on the completeness of the initial model. Since the initial model is obtained by traversing the graphical interface of the application using simple sampling strategies, complex GUIs seriously limit the effectiveness of these techniques. On the other hand, white-box approaches suffer from the complexity of the source code.

*Correspondence to: Leonardo Mariani, University of Milano Bicocca, viale Sarca 336, Milano, Italy.
†E-mail: mariani@disco.unimib.it

This paper presents AutoBlackTest, a black-box test generation technique that does not rely on an initial model but 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 by exploiting Q-learning [10] to turn the problem of generating test cases for an unknown application into the problem of learning how to act effectively in an unknown environment.

This paper extends previous work [11] by

- Defining a mechanism that reuses the information in existing test suites to improve the effectiveness of AutoBlackTest;
- Defining a mechanism for using test data based on the values expected by input widgets;
- Integrating and empirically investigating a new action selection policy;
- Empirically investigating the effectiveness of AutoBlackTest when different types of initial test suites are available.

The paper is organized as follows. Section 2 overviews the AutoBlackTest technique, provides the background information about Q-learning necessary to understand the rest of the paper and describes how Q-learning is integrated in AutoBlackTest. Section 3 describes the Q-learning infrastructure that is composed of the Observer, the Behavioural Model, the Executor, the Learner and the Planner. Section 4 describes how to use existing test suites to initialize the Q-learning model and improve the effectiveness of AutoBlackTest. Section 5 describes how to synthesize the concrete test cases that can be reexecuted by testers over time and discusses the types of failures that can be revealed with AutoBlackTest. Section 6 presents a prototype implementation of AutoBlackTest and discusses the empirical results obtained with the prototype implementation on various case studies. Section 7 discusses related work. Section 8 summarizes the main contributions of the paper.

2. AUTOBLACKTEST

AutoBlackTest is a testing technique that automatically generates test cases for interactive applications. An example of interactive application is the Universal Password Manager (UPM)[‡], an application for storing and handling user names, passwords, URLs and generic notes in a local or remote database. UPM interacts with the users through a main window that opens at the start-up of the application, and a few additional windows designed to handle specific situations. AutoBlackTest can generate system test cases for UPM by simulating user interactions. For instance, AutoBlackTest can automatically generate a test that creates a new account on UPM by clicking on the button that opens the window for the creation of new accounts, entering some text in the fields displayed in the newly opened window and pressing the ok button. UPM will be used as a running example through the paper, since it represents well the target application domain of AutoBlackTest and is simple enough to be easily discussed.

AutoBlackTest generates test cases by means of a component called Test Case Selector that elaborates the output of a Q-learning agent [10]. Although AutoBlackTest can be applied without any a priori knowledge of the system under test, its effectiveness can be improved by providing an initial test suite with test cases that indicate how to use the most relevant functions of the application. Figure 1 shows the main components of AutoBlackTest.

The *Q-learning agent* interacts with the application under test to identify and pursue executions that result in relevant computations, such as computations that add, modify and eliminate accounts in UPM. 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 this information in a model that the Q-earning agent exploits to decide the next action to execute. The model represents the knowledge about the application that has been acquired by the agent and is incrementally extended during the testing process. The Q-learning agent executes a fixed number of actions before starting a new sequence of actions from a random state,

[‡]http://upm.sourceforge.net/.

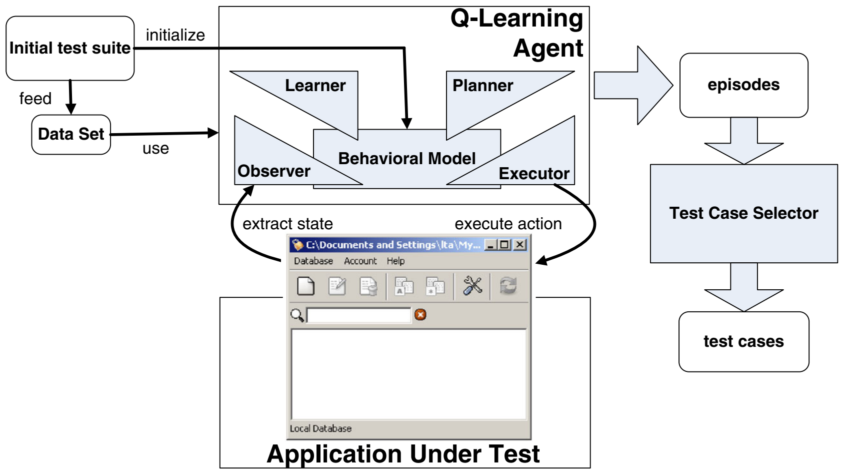


Figure 1. The overall structure of AutoBlackTest.

selected from the ones that have been already visited by the agent. The sequence of actions executed before starting a new one is called an *episode*.

The behaviour of the Q-learning agent is disciplined by two main parameters: the length of the episodes and the duration of the testing process. The length of the episodes is the number of actions that the Q-learning agent executes before stopping an episode and starting a new one. The duration of the testing process is the time that the agent allocates to explore the interface and build the model. At the end of the exploration, the *Test Case Selector* synthesizes a test suite, which includes a small and nonredundant set of episodes that can be used to validate future revisions of the application under test.

When test suites designed by testers are available, AutoBlackTest takes advantage of the available test cases in two ways. First, it executes the test cases to produce an initial model of the application. This initial model includes states and interactions that are relevant according to the testers' viewpoint. AutoBlackTest uses this model as a starting point for generating test cases. In this way, many relevant states and interactions do not need to be discovered because they are already in the model, and AutoBlackTest can focus its activity on generating relevant variations of the cases already sampled by the test suite. Second, AutoBlackTest populates its data set with the input values that occur in the test suite. This strategy produces a data set with values that are meaningful for the application under test. During testing, AutoBlackTest uses the values in the data set to fill input widgets. When an initial test suite is not provided, AutoBlackTest initializes the data set with a set of default values manually specified by the tester.

2.1. *Q-learning in a nutshell*

Q-learning [12] is a well-established reinforcement learning technique in which an agent learns how to act optimally in an environment through trial-and-error interactions. At each interaction, the agent performs an action, based on the current state of the environment, and evaluates the consequences of that action in terms of its immediate reward. The overall goal is learning how to act in a way that maximizes the cumulative reward, that is maximize the reward that is collected when executing an entire sequence of actions.

More formally, let S and A be the set of the possible states of the environment and the set of the actions that can be executed by the agent, respectively. In Q-learning, the agent interacts with the environment at some discrete time scale $t = 0, 1, 2, \ldots$. At each time step t, the agent monitors the current state of the environment $s_t \in S$, uses this state information to choose an action $a_t \in A$ and executes it. As a consequence, the environment reaches a new state and the agent collects a reward. The reached state and the reward depend only on the previous state and the executed action. The state transition function is represented with the function $\delta : S \times A \rightarrow S$, and the reward function is represented with the function $reward : S \times A \rightarrow \mathbb{R}$. Executing an action a_t in the state s_t

brings the environment in a new state $s_{t+1} = \delta(s_t, a_t)$ and collects a reward $r_t = reward\,(s_t, a_t)$. The transition function δ and the reward function *reward* characterize the environment and are not necessarily known to the agent.

The task of the agent is to learn an action-value function $Q : S \times A \rightarrow \mathbb{R}$ that returns, for any combination of state s_t and action a_t, the best cumulative reward that can be achieved by executing a sequence of actions that starts with a_t from s_t. The complete knowledge of the Q function can be exploited by the agent to behave optimally, by executing the actions that return the best cumulative reward.

The Q-function is recursively defined as follows:

$$Q(s_t, a_t) = reward(s_t, a_t) + \gamma \max_{a_{t+1}} Q(\delta(s_t, a_t), a_{t+1}) \tag{1}$$

where the discount-rate 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 rewards: a value close to 1 assigns higher weight to future rewards, while a value close to 0 assigns higher weight to immediate rewards.

The recursive definition of Q provides the basis for an algorithm that iteratively approximates it. In particular, the Q function is learnt incrementally based on the experience of the agent. The initial Q-value for each state-action pair is assigned with a user-defined default value. Then every time the agent executes an action a_t from a state s_t, the Q-value is updated according to the following rule:

$$\hat{Q}(s_t, a_t) \leftarrow \hat{Q}(s_t, a_t) + \alpha \left[r_t + \gamma \max_{a} \hat{Q}(s_{t+1}, a) - \hat{Q}(s_t, a_t) \right] \tag{2}$$

where the symbol $\hat{Q}$ indicates the current estimate of the actual Q-function, α is the learning rate, which is a real value in the range $(0, 1]$ that represents how strongly the new observation must affect the estimated Q-values, s_{t+1} is the state reached after the execution of a_t and r_t is the reward obtained with the execution of a_t.

Even though the equation (1) defines Q in terms of the functions $\delta(s, a)$ and $r(s, a)$, the agent does not require to know these functions to apply the updating rule (2). In fact the updating rule is based only on the observation (i.e. on the sampling) of the traversed states s_t and s_{t+1} and the immediate reward r_t.

To incrementally update the estimate of the actual Q-function, the agent must interact with the environment according to a policy. In principle, the agent could always randomly select the action to be executed among the ones that can be executed from the current state. In the long term, this policy results in a uniform exploration of the execution space, but with very slow convergence. An alternative approach consists of exploiting the estimated Q-values by always executing the action with the highest Q-value. However, this policy may restrict the exploration to the actions that are found during the early activity of the agent, failing to explore other actions that could have even higher Q-values. For this reason, the common policies adopted in Q-learning are based on probabilistic approaches: the actions with higher Q-values are assigned with higher probabilities, so as to exploit what the agent has already learned, but every action is anyway assigned with a nonzero probability, to keep exploring alternative actions.

The Q-learning algorithm is guaranteed to converge to the true Q-function if applied to a Markovian environment, with a bounded immediate reward and with state-action pairs continually updated [13].

The interested reader can refer to the studies by Sutton *et al.* [10] and Watkins [12] for additional details on Q-learning.

2.2. *Q-learning agent*

The Q-learning agent is an instance of Q-learning [10] where the Observer, the Learner, the Planner and the Executor have been designed to address the problem of generating test cases for interactive applications. The architecture of the Q-learning agent matches the architecture of a classic autonomic component [14], as shown in Figure 1.

The Q-learning agent executes episodes sequentially. When executing a new episode, the Q-learning agent randomly selects a state from the visited ones and starts executing the episode from the selected state (the states represent the GUI states of the application and are incrementally included into the model; the first episode is executed from the initial state of the application). If an initial test suite is available, the choice of the state to start from is restricted to one of the states visited while executing the initial test suite. The intuition is that the set of states visited while executing the initial test suite should be preferred because they include many relevant states that cover important behaviours of the application, at least according to the rationale of the testers who generated the test suite, while the distribution of the states dynamically discovered by AutoBlackTest can be biased by the Q-learning process.

The Q-learning agent brings the application into the selected state by executing the shortest sequence of actions that has been executed in any of the previous episodes to reach that state. Since the selected state is in the model, it is guaranteed that at least one sequence exists. The behaviour of the application along the executed sequence may be nondeterministic. If the sequence does not bring the application into the desired state, AutoBlackTest starts the episode from the reached state.

Starting the execution of an episode from one of the visited states is a standard assumption of Q-learning algorithms, and it is useful to increase the likelihood of exploring the whole space, in our case the execution space of the application under test. In fact, if every episode is started from the initial state, the agent could not reach the states that require the execution of a number of actions greater than the maximum length of an episode.

The Q-learning agent iteratively determines an 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 the collection of the property values of the GUI widgets of the applications. In *step 2*, the *Learner* updates the model according to (i) the state reached during the execution, (ii) the action that has been executed and (iii) the immediate utility of the action, accordingly to the formula (2). In *step 3*, the *Planner* applies a policy, based on the behavioural 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.

Under specific conditions, the Q-learning process converges to an optimal solution [13]. In our case, the condition that each state-action pair is continually visited does not hold, since the Q-learning agent can only explore (i.e. test) a portion of the entire execution space of an application due to the limited availability of the testing time. Thus, this paper aims to define an effective short-term exploration strategy, without focusing on convergence. The empirical results presented in Section 6.2 demonstrate that AutoBlackTest is effective despite the lack of convergence.

3. Q-LEARNING AGENT INFRASTRUCTURE

This section discusses in detail the main components of the Q-learning agent that is composed of the Observer, the Learner, the Behavioural Model, the Learner and the Planner.

3.1. Observer

The Observer accesses the application under test to build an abstract representation of its actual state. Ideally, it would like to precisely represent the state of the application, but in practice, it is hard to access the internal state of an application; even when it is possible to access it, it is usually too big and complex to be handled effectively. 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. The state extracted by the Observer is used as part of the model maintained by the Learner.

The Observer implements a function that takes a concrete GUI state as input and produces an abstract state as output. A concrete GUI state 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 $P^i = \left\{p_1^i, \dots, p_{n_i}^i\right\}$ is a set of properties. Each property $p_j^i = \left\{n_j^i, v_j^i\right\}$ is a pair, where n_j^i is the name of the property and v_j^i is the value of the property.

Each widget is usually associated with a large number of properties and including all the properties in the state information would produce huge state representations. The Observer reduces the size of the state by means of an abstraction function prj that transforms a concrete state S into an abstract state AS, $AS = prj(S)$. The function prj is a projection that is applied to every widget in the state S, $prj(S) = \{prj(w_1), \ldots, prj(w_n)\}$. Given a widget $w_i = (type_i, P^i)$, prj extracts a subset of its properties, that is $prj(w_i) = (type_i, P'_i)$, where $P'_i \subseteq P_i$. For each widget w_i, the subset of properties P'_i is selected according to the widget type $type_i$.

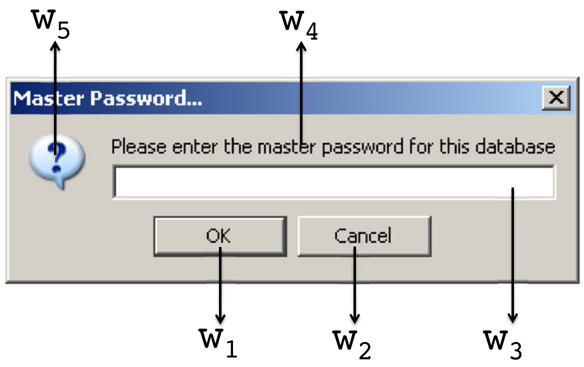
Table I shows a sample GUI state of the UPM application (top) and the corresponding abstract state computed by the Observer (bottom). The abstract state consists of the five widgets (W1-W5) that belong to the UPM window and a subset of their properties. The prj function extracts from each widget a set of properties that characterize the widget type.

For instance, the projection function selects the properties `uIClassID`, `text` and `editable` for the Password field (widget W3) that belongs to the UPM window shown in Table I. AutoBlackTest currently supports about 40 types of widgets and covers most of the elements that are incorporated in a GUI. In the infrequent case of an unsupported widget, AutoBlackTest extracts a standard set of properties that includes `class` and `uIClassID`.

The abstraction mechanism reduces the size of the model that the agent can explore by ignoring the widget properties that are considered irrelevant to represent the behaviour of the application. For instance, the abstraction mechanism ignores the property that specifies the color of a button because

Table I. An example of state abstraction of the UPM application.

Concrete GUI State



Abstract State

	widget type	property name	property value

W1	BUTTON	`uIClassID`
		`text`
		`enabled`
			`ButtonUI`
			`OK`
			`true`

W2	BUTTON	`uIClassID`
		`text`
		`enabled`
			`ButtonUI`
			`Cancel`
			`true`

W3	PASSWORD FIELD	`uIClassID`
		`text`
		`editable`
			`PasswordFieldUI`
			`null`
			`true`

W4	LABEL	`uIClassID`
		`text`
			`LabelUI`
			`Please enter the master for this database`

W5	LABEL	`uIClassID` `text`	`LabelUI` `null`

this property is both likely constant within a testing session and unlikely to influence the behaviour of the application.

The fact that a same abstract state may correspond to multiple concrete states does not represent an issue for AutoBlackTest as long as the concrete states react to the actions executed by the agent in the same way. The case of two concrete states with a different behaviour that are mapped to a same abstract state may introduce a local instability of the Q-values. This local phenomenon does not represent a problem for the overall exploration of the application, unless these cases are extremely frequent in the model. In this worst case, the effectiveness of AutoBlackTest is roughly reduced to the effectiveness of a random exploration of the application. Although this case is theoretically possible, in the practice it may happen only for the applications that exploit in an unusual way many of the properties in their widgets. This case has never occurred in the experiments carried out in this study.

Besides 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 behavioural model. To detect multiple occurrences of the same widget, AutoBlackTest uses 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. More formally, given a widget $w_i = (type_i, P^i)$, $trait$ extracts a subset of its properties, that is $trait(w_i) = (type_i, P_i')$, where $P_i' \subseteq P_i$. For each widget w_i, the subset of the properties P_i' is selected according to the widget type $type_i$. Traits are used to recognize a same widget even if some of its characterizing properties are changed. For instance, the trait of a button includes its type (for instance, `Button`), the position of the button in the GUI hierarchy and the label visualized on the button (for instance ‘OK’). Given two widgets w_1 and w_2, the equality $w_1 =_t w_2$ is valid iff $trait(w_1) = trait(w_2)$. This strategy is similar to a feature offered by IBM Functional Tester to compare widgets [15]. Based on this definition, the following restriction operator between abstract states that will be used in the next sections can be defined: given $AS = \{w_1, \dots, w_n\}$, $AS' = \{w_1', \dots, w_{n'}'\}$ abstract states, $AS \setminus_t AS' = \{w_i | w_i \in AS \wedge \nexists w_k' \in AS' s.t.\ w_i =_t w_k'\}$.

3.2. Learner and behavioural model

The Learner builds and updates the behavioural model according to the activity of the Agent. The behavioural model is defined as a tuple that incorporates the result of Q-learning. More formally, the behavioural model is a tuple $(AS, A, \delta, reward, Q)$ where AS is the set of the abstract GUI states of the application returned by the Observer, A is the set of actions that can be executed on the application, $\delta : AS \times A \to AS$ is the state transition function, $reward : AS \times A \to \mathbb{R}$ is the reward function and $Q : AS \times A \to \mathbb{R}$ is the estimated Q-function.

The behavioural model can be visually represented with a multidigraph that is a graph where pairs of nodes can be connected by multiple directed edges. Nodes represent the abstract states AS, while edges represent actions in A and connect nodes according to the transition function δ. Edges are annotated with the values of the reward and the Q-value, returned by the *reward* and Q functions, respectively. Every time the Agent observes a new state or executes a new action, it extends the behavioural model accordingly.

The Q-value that annotates edges represents the likelihood that the corresponding transition causes or enables complex interactions with the application, which can be extremely useful for system testing. As usual in the Q-learning process, the Learner assigns a Q-value to a transition after executing the action that corresponds to the edge. The Learner computes the Q-value according to both an immediate utility value and the Q-values of the actions associated with the edges that exit the state reached with the current edge (see formula 2 in Section 2.1).

In the following, the reward function that computes the immediate utility of an action is first presented, and then the computation of the Q-values is discussed.

The value of actions with respect to testing depends on the computations activated by the actions. The reward function favours 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.

For example, the form shown at the top of Figure 2 is the form for creating new accounts in UPM. If AutoBlackTest clicks the `OK` button, a new account is created and a new window is displayed (left-bottom corner of Figure 2). Intuitively, the major change on the displayed widgets corresponds to an execution that is immediately relevant for testing: a new account has been created. On the contrary, if AutoBlackTest clicks the `Hide` check box, the new GUI state is only marginally different from the previous state (right-bottom corner of Figure 2). Intuitively, the small difference in the GUI state corresponds to an interaction that produced an execution with small immediate relevance for testing.

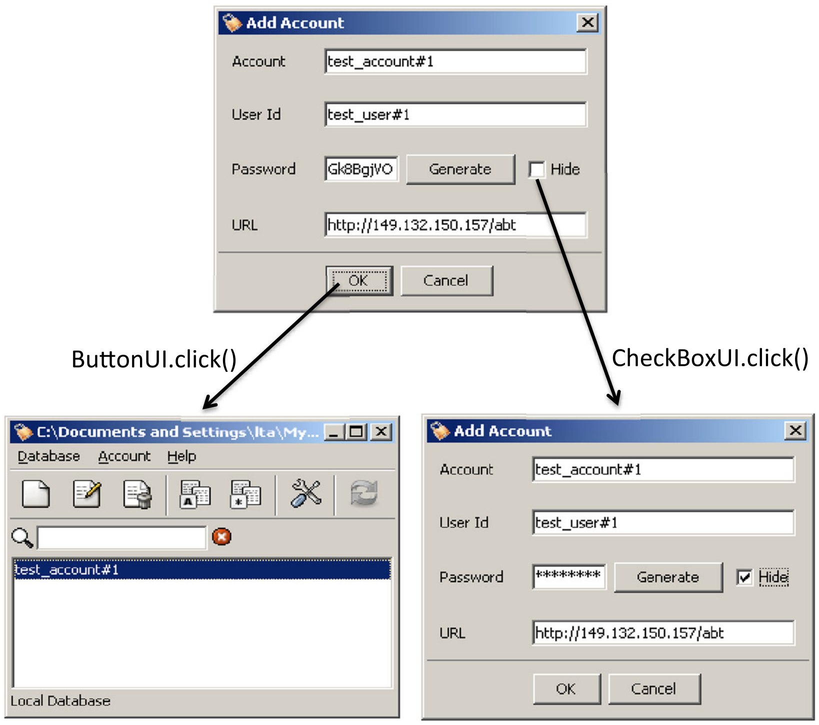
The heuristic is effective but not perfect. However, the randomness in the exploration process that starts episodes from random states and executes random actions frequently prevents the agent from wasting resources in uninteresting areas of the execution space, due to exceptional cases that violate the heuristic.

To define the reward function, the diff_w function is introduced. This function 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 $w_1 = (type, P_1)$ and $w_2 = (type, P_2)$, such that $w_1 =_t w_2$, the diff_w function computes the proportion of properties that changed their value:

$$\mathit{diff}_w(w_1, w_2) = \frac{|P_1 \setminus P_2| + |P_2 \setminus P_1|}{|P_1| + |P_2|} \tag{3}$$

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:



Account: test_account#1
User Id: test_user#1
Password: Gk8BgjVO
URL: http://149.132.150.157/abt

test_account#1

Account: test_account#1
User Id: test_user#1
Password: ********
URL: http://149.132.150.157/abt

Figure 2. An example of actions of the UPM application.

$$diff_{AS}(AS_1, AS_2) = \frac{|AS_2 \setminus_t AS_1| + \sum_{w_1 \in AS_1, w_2 \in AS_2, w_1 =_t w_2} diff_w(w_1, w_2)}{|AS_2|} \tag{4}$$

The reward function is now defined. Given an abstract state AS_1 and an action a executed from AS_1, the state reached by executing a from AS_1 is defined with $\delta(AS_1, a)$. The reward of the action is the proportion of new widgets and widgets that changed their properties from the original to the target state: $reward(AS_1, a) = diff_{AS}(AS_1, \delta(AS_1, a))$. The widgets that disappear when moving from the original to the target state are not considered to avoid incrementing the Q-values too quickly when actions change the windows. When the increments of Q-values are too fast, the activity of the agent tends to focus too much on the actions that cause the big increments, in this case transitions between windows, ignoring the windows themselves. In the context of testing, this behaviour corresponds to spending more time in testing transitions between windows rather than testing functionalities. Moderate increases of the Q-values lead to better exploring the state space and thus to a testing activity that balances transitions between windows with executing functionalities.

The reward function defined above can estimate well the immediate utility of an action but does not take into account the utility of that action when executed as a part of a sequence of actions. Sometimes 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 reward function and the utility of the actions that can be executed in the future [10]. Considering the reward of future actions allows the agent to effectively go explore situations that require executing some preliminary actions to enable the execution of the action relevant for testing. At each step, the Planner uses the Q-values to choose the action to be executed.

The role of the Q-values is exemplified in the model shown in Figure 3. The model is a simplified excerpt of the model of the UPM application. For the sake of simplicity in the figure, the states are labelled with the name of the UPM windows (instead of the 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 the parameters).

The path through the states 00, 01, 10, 11, 02, 03, 14 represents the case of a user who tries to import a remote database of passwords by (i) selecting the `Open Database From URL` command from a menu, (ii) setting the address to connect to the remote database from the `ConnectionSetting` window, (iii) clicking on an existing database in the `FileChooser` window, (iv) clicking the `Save` button and (v) confirming the intention to overwrite the existing database. The path reaches an error state because it corresponds to an attempt to connect to a remote database without specifying a correct communication protocol. The edges are labelled with (i) the action that triggers the transition between the states, (ii) the Q-value (`Q:`) and (iii) the Reward value (`R:`) of the corresponding action. The transitions with bold labels indicate that the Learner can assign high Q-values both to actions with high reward value (for instance, selecting the `File` menu, which enables interesting scenarios) and to actions with low reward value (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.doubleClick()` in Figure 3 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.

The Learner computes Q-values starting from the reward values according to the standard Q-learning formula for $\alpha = 1$ and $\gamma = 0.9$. Since the learning rate α determines the impact of the new observations on the model and AutoBlackTest needs to quickly learns how the tested application behaves, the value 1 is selected to produce the greatest impact on the model at each interaction. Since the discount factor γ balances the relevance of the immediate reward with respect to future actions

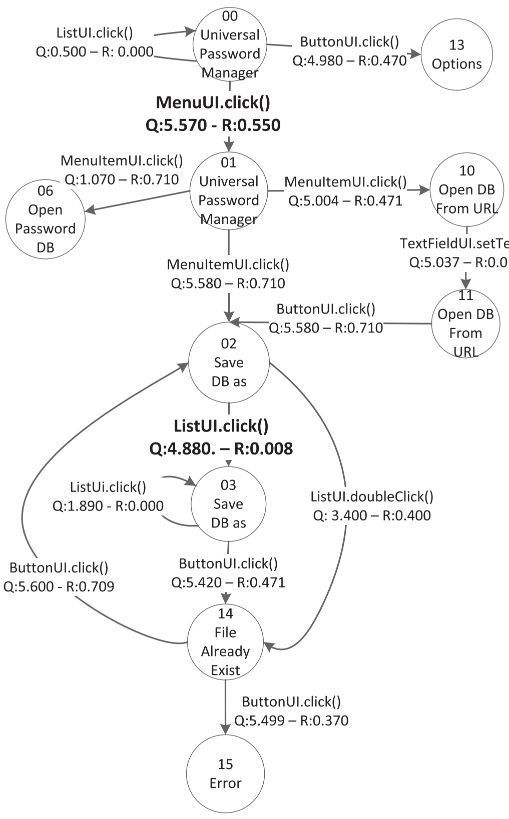

00 Universal Password Manager
ListUI.click() Q:0.500 – R: 0.000
ButtonUI.click() Q:4.980 – R:0.470
13 Options
MenuUI.click() Q:5.570 - R:0.550
01 Universal Password Manager
MenuItemUI.click() Q:1.070 – R:0.710
06 Open Password DB
MenuItemUI.click() Q:5.004 – R:0.471
10 Open DB From URL
TextFieldUI.setText() Q:5.037 – R:0.015
11 Open DB From URL
MenuItemUI.click() Q:5.580 – R:0.710
ButtonUI.click() Q:5.580 – R:0.710
02 Save DB as
ListUI.click() Q:4.880. – R:0.008
03 Save DB as
ListUi.click() Q:1.890 - R:0.000
ListUI.doubleClick() Q: 3.400 – R:0.400
ButtonUI.click() Q:5.600 - R:0.709
ButtonUI.click() Q:5.420 – R:0.471
14 File Already Exist
ButtonUI.click() Q:5.499 – R:0.370
15 Error

Figure 3. Excerpt of a model derived with AutoBlackTest.

and AutoBlackTest aims to execute sequences of actions that maximize the reward collected during an entire episode, rather than maximizing the immediate reward, the value 0.9 has been selected. Other studies in Reinforcement Learning [16, 17] confirm that $\gamma = 0.9$ produces highly effective results. The resulting formula is

$$Q(s,a) = reward(s,a) + 0.9 \max_{a'} Q(\delta(s,a), a') \tag{5}$$

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 $\delta(s,a)$ and *reward(s,a)* indicates the immediate reward of the action a executed from the state s.

When an action reaches a new state, the second term of the formula is zero and the Q-value is the reward value. Otherwise, the Q-value is computed according to both the reward value 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.

3.3. Planner and executor

The *Planner* selects the action executed at each iteration according to a policy. In AutoBlackTest, two popular policies used in Q-learning were considered: the *ϵ-greedy* [16, 17] and the *softmax* policies [10].

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 behavioural model available to the agent, with probability $1 - \epsilon$. The value chosen for ϵ can impact significantly the effectiveness of the technique.

The softmax strategy consists of choosing the action according to a probability distribution that depends on the Q-value of the actions. The softmax strategy depends on a parameter that influences the probability distribution. This parameter can be assigned with a value in the range [0..1], where low values of the parameter strongly favour the actions with high Q-values over the action with medium or low Q-values, while high values of the parameters result in a uniform probability distribution, regardless of the distribution of the Q-values.

A suitable policy (either ϵ-greedy or softmax) and a suitable value for its parameter were determined by empirically comparing the performance of AutoBlackTest when using the two policies under different configurations. The empirical study reported in Section 6.2 shows that the 0.8-greedy policy produces the best results.

The *Planner* distinguishes between simple and complex actions. A *simple action* is a single event triggered on a widget. A *complex action* is a workflow of simple actions orchestrated heuristically. Complex actions are defined to face *specific situations* effectively.

For instance, if the same window displays several widgets that accept data values, the planner can select the complex action `fillForm` that fills out all or most of these widgets before clicking a button. The rationale of the action is that to process the data entered through the widgets that comprise the form is likely necessary to fill out most of them; otherwise, the data entered in the window are likely to be insufficient. In principle, it is possible to obtain the same result combining several simple actions, but the time needed to learn how to fill the widgets and then clicking on the submit button may be undesirably long.

A complex action is characterized by an *enabling condition* that specifies the condition under which the complex action can be executed and that consists of a set of constraints on the number, type and properties of the widgets that must occur in the windows. If all the constraints evaluate to *true*, the Planner can select the complex action.

When the *Planner* selects a random action, it behaves differently depending on the available actions and on the availability of an initial test suite. If an initial test suite is available, the *Planner* uses only simple actions, since the sequences of actions necessary to interact with complex windows, like in the case of a complex form with multiple dependencies between input widgets, are already available to the agent in the form of the model produced from the initial test suite. Thus, AutoBlackTest mainly produces variations of existing cases and seldom faces a completely new situation. If no initial test suite is available, every window that is accessed for the first time is a new situation for AutoBlackTest, and thus, AutoBlackTest considers both simple and complex actions. When both simple and complex actions are executable in the current state, the *Planner* selects a complex one with a probability of 75%, as empirically determined through experiments.

The *Executor* interacts concretely with the widgets and is activated to execute either a simple or a complex action. In the following, the simple and complex actions currently supported by AutoBlackTest are described.

Simple actions. A simple action is a single action executed on a single widget. Section 6.1 specifies the set of supported widgets. Sometime actions require a parameter to be executed. For instance, a `textarea` requires a text to be entered.

AutoBlackTest handles parameter values by exploiting a catalog of predefined literals that can be used for testing. In particular, AutoBlackTest analyzes the GUI and automatically associates a label that occurs in the GUI to each input widget available in the current window using the algorithm described by Becce *et al.*[18]. The association rule is straightforward and takes advantage of the GUI design principles. In most of the cases, it consists of associating an input widget with the closest label that occurs in the GUI. For instance, AutoBlackTest can discover that `name`, `birthday` and `address` are the labels that describe the data expected in three text areas.

The catalog of predefined literals that can be used to support testing is organized according to a set of user-defined labels. The catalog may contain names, dates, addresses and so on. When AutoBlackTest needs a parameter value for a simple action, it first checks if the label that describes the input widget belongs to the catalog. If it does, AutoBlackTest randomly selects a value from the set of values associated with the label found in the catalog. If the label does not occur in the catalog, AutoBlackTest uses the predefined string `genericstring`.

The catalog can be both manually populated with legal, boundary, illegal and special values, and automatically populated by reusing values from existing test cases. Manually populating the catalog is an interesting option for the testers who can specify the data that AutoBlackTest must use to test a specific application. The effort necessary to manually populate a catalog depends on the testers' experience and their knowledge of the application under test.

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 behaviour of the action. When a complex action executes an action that requires a parameter, it exploits the catalog of predefined literals available to simple actions.

4. INITIAL TEST SUITE

AutoBlackTest can be used both in presence and absence of test cases. When test cases are available, AutoBlackTest takes advantage of them by reusing both the parameter values and the test sequences.

When reusing the parameter values, AutoBlackTest scans the test cases, extracts the input values defined by the test designers and stores these values in a catalog of values that is used to generate new test cases. When storing values, AutoBlackTest takes advantage of its ability to associate labels with widgets to classify the parameter values. For instance, if a test case enters the literal `John` in a textarea associated with a label `name` and the literal `New York` in a textarea associated with a label `city`, AutoBlackTest classifies the literal `John` as a name and the literal `New York` as a city and reuses them only with widgets associated with labels `name` and `city`, respectively. The reuse of parameter values is a way of feeding the catalog with literals specifically meaningful for the application under test.

The reuse of test sequences consists of computing an initial model that is used as starting point for the AutoBlackTest testing activity. AutoBlackTest computes this initial model by executing the test suite and monitoring the program execution.

The execution of each test case produces a model with the states that have been visited, the actions that have been executed and the Q-values as the reward computed for each executed action of the discovered states. As an example, Figure 4(a) shows the models that AutoBlackTest computes when executing two test cases for the UPM application.

Executing a valuable and highly rewarded functionality (e.g. submitting a request) often requires executing several actions with a small reward (e.g. filling the many fields of a form). For instance, the `TextFieldUI.setText()` action from state 2 to state 3 in the test TC1 shown in Figure 4(a) defines the name of the newly created database and has a small reward (0.008). However, such an action is necessary to successfully execute the next action `ButtonUI.click()` that saves the database and has a high reward (0.471). The different reward of the actions is reflected in the Q-values.

The occurrence of highly valuable actions does not impact on the Q-value of the actions executed earlier in the test. This is due to the lack of propagation of the Q-values that in Q-learning happens when the same actions are executed multiple times. To maximize the benefits of the available test cases, the information about highly rewarded actions are propagated up to the root of the test case, so that the model can be exploited to take decisions that are effective on the long-term execution of the test cases.

The Q-values are propagated by updating the Q-values on every action of the test as many times as the length of the test (i.e. the number of actions that have been executed). In this way, the reward produced by every action influences the Q-value of every other action in the model, in a way that is proportional to the distance between actions. The propagation strategy applied to the test cases in Figure 4(a) returns the models in Figure 4(b). In the new models, the rewards of late actions influence the Q-values of the early actions. For instance, the Q-value of the aforementioned `textFieldUI.setText()` action increases from 0.008 to 0.681.

AutoBlackTest produces the initial model by merging the models of the single test cases. The merging is straightforward: the states of the initial model are the union of the states of the test models, and the transitions are mapped from the test models to the initial model consistently. There

Table II. Complex actions currently supported in AutoBlackTest.

Action name	Situation	Enabling condition	Behaviour
File Chooser	This action is designed to interact with an open/save window (which is a standard modal window provided by most GUI frameworks). The action aims to limit the scope of the interaction with the file system and produce legal file names.	The action detects the open/save window 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 before completing the action and requires to enter filenames from a predefined set of valid filenames.
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 action aims to support the particular grid that is displayed in this kind of window, and that would not be usable with simple actions.	The action detects the color choosing window 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 centre 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, closes the new window first and 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.	The action identifies a form-like window by checking if at least eight widgets that allow entering data (widgets that consist of `textArea`, `comboBoxes`, `Trees`, etc.) and a button are active in the current window.	This action can have six behaviours obtained by combining the value of two parameters. The first parameter represents the percentage of the widgets that allow entering data to be filled and 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 needed to select multiple items in listboxes.	This action is enabled if a listbox that allows selecting multiple items is active in the current window.	This action can have three behaviours. It can select two, half of 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.	This action is activated if at least 3 widgets of the same kind are active in the current window.	This action can have three behaviours. It can fill 50%, 75% or 100% of the widgets.

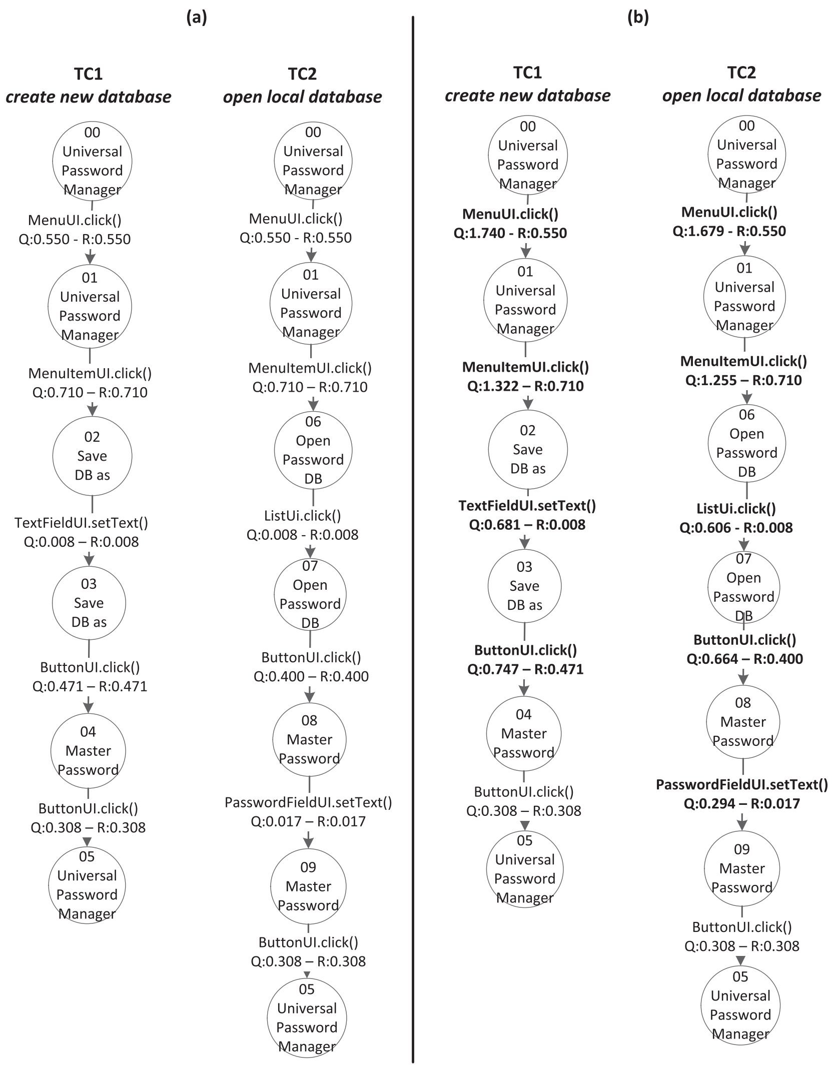


Figure 4. (a) The models produced by executing two test cases of the UPM application. (b) The models obtained after propagating the Q-values.

is only one case of possible conflict that occurs when a same action from a same state occurs in two different test models with different Q-values. In this case, the initial model include the action with the maximum Q-value, following the rationale that the maximum Q-value represents the best opportunity that exists in one of the possible continuations of the execution.

The sample test models in Figure 4(b) include a case of conflict. The action `MenuUI.click()` occurs from state 0 to state 1 in both test models but with different Q-values: 1.740 in TC1 and 1.679 in TC2. The different Q-values are originated by the different evolutions of the tests: one test creates a new database, while the other opens a local database. To represent the best opportunity that the agent has along the path that starts with `MenuUI.click()`, as required by Q-learning, the highest Q-value, 1.740, is included in the initial model. Figure 5 shows the initial model obtained from the test models in Figure 4(b).

5. TEST SYNTHESIS AND FAILURE DETECTION

This section discusses the synthesis of test cases and the failure detection capabilities of AutoBlackTest.

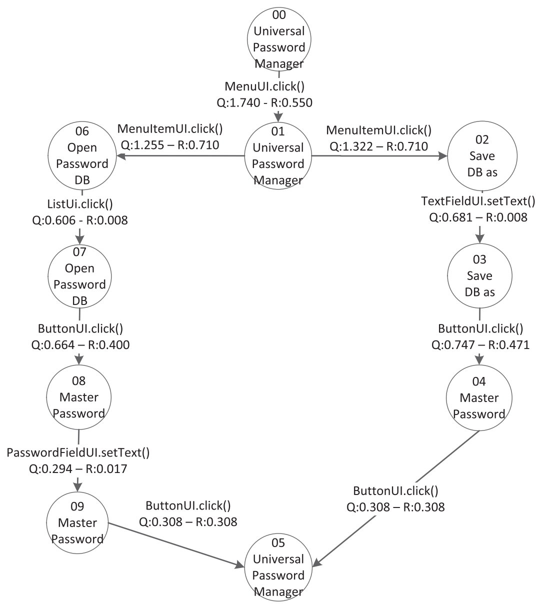

00 Universal Password Manager
MenuUI.click() Q:1.740 - R:0.550
01 Universal Password Manager
MenuItemUI.click() Q:1.255 – R:0.710
06 Open Password DB
MenuItemUI.click() Q:1.322 – R:0.710
02 Save DB as
ListUi.click() Q:0.606 - R:0.008
TextFieldUI.setText() Q:0.681 – R:0.008
07 Open Password DB
03 Save DB as
ButtonUI.click() Q:0.664 – R:0.400
ButtonUI.click() Q:0.747 – R:0.471
08 Master Password
04 Master Password
PasswordFieldUI.setText() Q:0.294 – R:0.017
09 Master Password
ButtonUI.click() Q:0.308 – R:0.308
ButtonUI.click() Q:0.308 – R:0.308
05 Universal Password Manager

Figure 5. The initial model obtained from the test models in Figure 4(b).

5.1. *Test case selector*

The episodes that AutoBlackTest executes automatically 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 test cases is straightforward. The current prototype implementation of AutoBlackTest generates test cases executable with IBM Functional Tester.

Episodes that execute the same sequences of actions are redundant. To produce a nonredundant test suite, the Test Case Selector filters the set of episodes following the *additional statement coverage prioritization* approach [19]: AutoBlackTest selects the episode with the highest statement coverage, adds it to the test suite, recomputes 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.

The test cases discarded according to this strategy might carry some useful information even if not contributing to the coverage. When storing a large test suite with a long execution time is acceptable, this step can be skipped and all the episodes can be stored as tests.

5.2. *Failure detection*

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. This section describes how domain independent failures can be detected.

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 not responding to any 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 to replicate the failing execution (if deterministic). Differently from the previous cases, it does not interrupt the execution of the episode but continues executing actions until the current episode is completed. If the application under test writes to log files, AutoBlackTest can be configured to monitor the content of the log files, in addition to standard output and error streams, and detect exceptions.

Regressions When the test cases produced by the Test Case Selector are reexecuted to validate a new version of the application under test, AutoBlackTest can use the failure detection mechanisms described by Xie *et al.* [20] that derive from the possibility of comparing the state information recorded the first time the test is executed with the state information observed during regression testing.

Xie *et al.* [20] shows that two effective oracles consist in (1) checking after the execution of each action whether the state of the active window matches the state of the active windows recorded in the test and (2) checking at the end of the test if the state of all the opened windows matches the states of the windows recorded at the end of the execution of the original test. AutoBlackTest can use both these oracles.

6. EXPERIMENTS

This section presents the prototype implementation of AutoBlackTest and the results of the empirical evaluation.

6.1. Prototype

The prototype implementation of AutoBlackTest integrates two third-party components: IBM Rational Functional Tester [15] and Teachingbox [21]. AutoBlackTest uses IBM Rational Functional Tester both in the Observer to extract the widgets of the application and in the Actuator to interact with the widgets. AutoBlackTest uses Teachingbox in the Learner to update the model of the application.

The current implementation supports Java/Swing but can be easily extended to any GUI framework supported by IBM Functional Tester, including JAVA, .NET and a number of other Windows and Linux GUI frameworks, with small changes in the Observer and the Actuator.

Table III summarizes the current 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.

The tool is available for download at http://www.lta.disco.unimib.it/tools/AutoBlackTest/ and requires IBM Rational Functional Tester to be installed and executed.

6.2. Empirical evaluation

AutoBlackTest was empirically evaluated by (1) studying the impact of the action selection policy on the effectiveness of the technique, (2) comparing AutoBlackTest with state of the art

Table III. Simple actions currently supported in AutoBlackTest.

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)

GUI testing techniques and (3) investigating the impact of the initial test suite on the effectiveness of AutoBlackTest.

6.3. *Action selection policy*

The impact of the action selection policy on the technique was investigated by using a small application with a simple GUI to limit the influence on the results of external factors that may be difficult to control in complex applications and keep the focus on the exploration strategies. The specific application that was selected is UPM v1.6 (2.515 locs), and the application used as running example. The effectiveness of the policies was evaluated by measuring the statement coverage obtained after 12 h of activity with no initial test suites. Statement coverage is used as a proxy measure of the exploration ability of the technique. High coverage would indicate that the technique can explore a large part of the execution space indeed–low coverage may depend on different factors–but can be interpreted as a symptom of potential problems. Each configuration was executed three times, and the effectiveness of the two policies described in Section 3.3 was empirically investigated: ϵ-greedy and softmax.

Since Q-learning is more effective when a significant proportion of random actions are executed, the effectiveness of the policies was investigated with values of ϵ and temp close to 1. For both policies, the parameter influences the degree of randomness in the decision about the action to execute; in both cases, when the parameter is equal to 1, the decision is purely random. The experiments were conducted with values 0.6, 0.8, 0.9 and 1 for each of the parameters.

Table IV shows the average statement coverage obtained with each configuration. The data indicate that the ϵ-greedy policy outperforms the softmax policy and the random selection of the action performs worst, while the other configurations perform well and reach the best results with $\epsilon = 0.8$ (in bold in the table).

Although these results are not final, the choice of the ϵ-greedy policy with a value of ϵ close to 0.8 is recommended when using AutoBlackTest. $\epsilon = 0.8$ is used as reference value for the rest of the experiments.

6.4. *Comparative evaluation*

The capability of supporting system testing is estimated by measuring the statement coverage obtained with AutoBlackTest, and by evaluating the ability of finding uncovered problems in the target application. The results are compared with GUITAR v1.1.1 that represents the state of the art in the field [6].

Four applications for desktop machines are selected as case studies for these experiments. Applications from different domains already exploited in similar studies (see, for instance, the GUITAR web page http://sourceforge.net/apps/mediawiki/guitar/) are chosen: UPM v1.6, a personal password manager (2.515 locs); PDFSAM v0.7 stable release 1 (http://sourceforge.net/projects/pdfsam/), a tool for merging and splitting PDF documents (3.138 locs); TimeSlotTracker v0.4 (http://sourceforge.net/projects/timeslottracker/), an advanced manager of personal tasks and activities (3.499 locs); and Buddi v.3.4.0.8(http://buddi.digitalcave.ca/), a personal finance and budgeting program (10.580 locs).

In this empirical study, the two techniques are assumed to be used in overnight sessions, and the results produced after 12 h of execution on an Intel i5 760 at 2.80 Ghz with 4GB ram are compared. Since AutoBlackTest generates test cases incrementally, its execution was simply interrupted after 12 h of execution time. Any initial test suite was not provided to AutoBlackTest.

Table IV. Statement coverage obtained with different policies.

policy	$\epsilon = 0.6$	$\epsilon = 0.8$	$\epsilon = 0.9$	$\epsilon = 1$
ϵ-greedy	84%	**87%**	84%	81%
policy	TEMP=0.6	TEMP=0.8	TEMP=0.9	TEMP=1
softmax	76%	80%	67%	81%

GUITAR can generate multiple models. GUITAR was configured 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 [5, 22]. A few testing sessions were run with different initial models to confirm that EIG is the best option.

GUITAR works by generating 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. Yuan *et al.* [5] suggest to use abstract test cases no longer than 10. To find an appropriate length for the experiment, GUITAR was applied to the case studies looking for configurations that terminates the test case generation in at most 6 h (50% of the budget). After executing GUITAR on the target applications, it is found that the maximum value that can be used to generate the test cases within the time limit for every application is 5 h. The remaining time, which has never been less than 6 h with an average of 10 h, is used 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 scores for these two case studies. To compare the two techniques, the applications are slightly modified to let GUITAR proceeds when facing these two windows and avoid getting stuck.

To reduce the impact of randomness in the experiments, AutoBlackTest and GUITAR were run for three times on every application, and average results are reported. The following two subsections report the results obtained for code coverage and fault detection.

Code coverage. Table V shows the statement coverage achieved in the experiments and the size of the test suite automatically generated with AutoBlackTest and GUITAR. Statement coverage is computed 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 behaviour 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 6 shows how AutoBlackTest incrementally increases statement coverage. As expected, the amount of covered statements increases fast with early 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. For example, AutoBlackTest can be terminated 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 regression test suites that can be reexecuted 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

Table V. Statement coverage achieved with AutoBlackTest and Guitar.

Application	Technique	Statement coverage	Number of 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

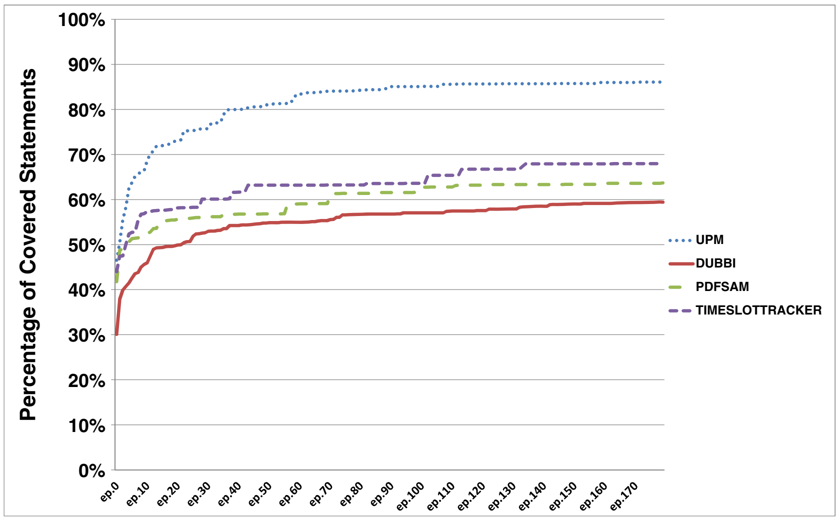


Figure 6. Average increment of statement coverage along episodes.

Table VI. Number of faults revealed by AutoBlackTest and GUITAR.

		AutoBlackTest faults			GUITAR faults		
Application	Execution	Severe	Minor	Total	Severe	Minor	Total
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)

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. Therefore, it is reasonable to 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).

Fault detection. The comparative evaluation of AutoBlackTest with GUITAR was completed by examining the faults that AutoBlackTest and GUITAR detected automatically while running with the configuration described above. Table VI shows the number of faults detected in each execution distinguishing between severe and minor faults. Faults are classified as severe if they prevent the execution of a functionality, minor otherwise. The average number of faults detected per execution and the total number of faults detected across the three executions (row *avg(tot)*) are also reported.

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. Moreover, 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 two applications: one fault in UPM and four in TimeSlotTracker. From this empirical validation, it is possible to conclude that AutoBlackTest is more effective than GUITAR in detecting faults when used in a 12 h time frame: AutoBlackTest detected a total of 20 faults, while GUITAR detected a total of five faults (four out of the five 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. It is particularly interesting the ability of AutoBlackTest to detect faults that can be revealed only through rare combinations of actions, as frequently happens for system level faults. This ability depends on the capability of AutoBlackTest to exploit the alternation of exploitation and exploration, which is typical of Q-learning. The readers should notice that the faults that can be activated only through the execution of multiple system-level actions, as the ones revealed by AutoBlackTest, can be hardly revealed with test cases designed to validate individual units, like unit test cases, which focus on functionalities implemented by a single unit, rather than on the integration of functionalities offered by multiple components of the system.

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.

In summary, AutoBlackTest effectively revealed faults that can be activated only by executing specific sequences of actions, sometime in combination with specific data values. This ability is clearly limited by the actions supported by AutoBlackTest and the values used to populate the data set.

6.5. *Impact of the initial test suite*

AutoBlackTest can take advantage of the availability of an initial test suite as discussed in Section 4. This section reports a study about the impact of different types of initial test suites on the effectiveness of AutoBlackTest. The study was conducted using the Jaolt 0.6.10 application[§], a desktop client for eBay auctions (19.677 locs). This application was selected because it includes several complex forms, which challenge AutoBlackTest, and it expected that some initial test suite could bring relevant benefits.

The system test suites for the study was manually designed, considering two main dimensions: the sequences of operations and the input values. Each dimension was reasoned in terms of positive cases, which are test cases that lead to the successful execution of the functionality targeted by the test case, and negative cases, which are test cases that include improper usages of the target functionality. According to these criteria, four artifacts were generated:

Dp a set of positive values obtained by defining a positive and correct input for each widget of the application;
Dn a set of negative values obtained by defining a negative and incorrect input for each widget of the application;
Tp a set of positive test cases, that is sequences of actions that successfully execute one or more functionalities of the application;
Tn a set of negative test cases, that is sequences of actions that are forbidden by the application.

The following six configurations were investigated for AutoBlackTest:

C_Dp The AutoBlackTest data set is populated with the values in Dp, simulating the case of a tester providing positive values to the technique;

[§]http://code.google.com/p/jaolt/.

Table VII. Statement coverage with different configurations and initial test suites.

Configuration	Initial coverage	Average coverage	Delta
Empty	0%	28.75%	28.75%
C_Dp	0%	28.52%	28.52%
C_DpDn	0%	31.06%	31.06%
C_TpDp	39.75%	44.68%	4.93%
C_TpDpDn	39.75%	43.34%	3.59%
C_TpTnDp	40.93%	44.5%	3.57%
C_TpTnDpDn	40.93%	43.51%	2.58%

C_DpDn The AutoBlackTest data set is populated with the values in Dp and Dn, simulating the case of a tester providing both positive and negative values to the technique;

C_TpDp AutoBlackTest uses the model obtained by executing Tp as initial model. The data set is populated with the data used in Tp, which matches the data included in Dp.

C_TpDpDn AutoBlackTest uses the model obtained by executing Tp as initial model. The data set is populated with both the data used in Tp, which matches the data included in Dp, and the data in Dn.

C_TpTnDp AutoBlackTest uses the model obtained by executing both Tp and Tn as initial model. The data set is populated with the data used in Tp, which matches the data included in Dp.

C_TpTnDpDn AutoBlackTest uses the model obtained by executing both Tp and Tn as initial model. The data set is populated with both the data used in Tp, which matches the data included in Dp, and the data in Dn.

The effectiveness of AutoBlackTest was measured by measuring both the statement coverage and the number of revealed faults. The results have been obtained by executing the technique three times in a 12 h session.

Table VII shows the statement coverage achieved in the executions. Column *Configuration* indicates the AutoBlackTest configuration: Empty indicates that AutoBlackTest executed without any additional information, neither about data nor about test sequences. Column *Initial Coverage* indicates the statement coverage obtained by executing the initial test suite only, when available. Column *Average Coverage* indicates the statement coverage obtained by executing AutoBlackTest. Column *Delta* indicates the increment of coverage produced by AutoBlackTest with respect to the initial test suite.

The results reported in the table indicates that AutoBlackTest with no additional information achieves the lowest statement coverage, still inspecting a relevant portion of the execution space autonomously. In fact, it covers more than 28% of the statements of the application, while the best manual test suites covered 40.93% of the statements.

The use of positive and negative values in the data set does not bring relevant benefits. In particular, the use of positive values does not impact significantly on the coverage, while the use of negative values improves the coverage from 28.75% (Configuration = Empty) to 31.06% (Configuration = C_DpDn), suggesting that introducing a notion of incorrect values in the technique can trigger some corner cases that AutoBlackTest can hardly cover otherwise.

With an initial test suite, AutoBlackTest increases the statement coverage up to 44.68%. The initial test suite that consisted of a fairly complete set of executions that samples analytically every functionality of the application, provided a coverage of about 40%, suggesting the possible presence of many statements unreachable in the configuration used in the experiment. The application code was inspected, and it was observed that the dead code and the code that implements functionality that it was not considered in the test[¶] amount for at least 20% of the code. This estimation is conservative, and the set of unreachable statements may be much larger. AutoBlackTest has been able to increase

[¶]Since jaolt is an eBay client, this study does not consider the functionality related to interactions with the concrete eBay service.

the code coverage automatically by an amount between 2.58% and 4.93%. Considering that these extra statements are covered automatically by identifying unusual combinations of functionalities that were already tested in the initial test suite and that the application contains a big amount of unreachable code, the result is encouraging. The relevance of this extra coverage is confirmed by the data about faults that indicate that AutoBlackTest discovered several real faults.

Table VIII reports the number of faults found with the different configurations. It indicates the AutoBlackTest configuration (column *Configuration*), the average number of faults revealed per testing session (column *Average faults*) and the total number of faults (column *Total faults*) distinguishing between severe (column *Severe*) and minor faults (column *Minor*).

The data reported in the table show that AutoBlackTest has been able to reveal faults in every configuration, strengthening the good results reported in the previous section. In total, six new faults were discovered, one severe and five minor ones. Figure 7 shows the faults discovered in each configuration.

By comparing the results obtained with only data values with the results obtained with an initial test suite, it is noticeable that with an initial test suite AutoBlackTest reveals a higher number of faults. In fact, AutoBlackTest revealed two faults only in the configurations C_Dp and C_DpDn, and three to four faults in the other configurations.

By comparing the results about the incremental coverage and the number of revealed problems, it is possible to confirm the intuition that even if the unusual combinations automatically generated by AutoBlackTest lead to a small increment of the coverage, the extra combinations tested in this way reveal several faults. Small increments in statement coverage (between 2.58% and 4.93%) can reveal more faults (between 2 and 4).

A surprising aspect is that with the Empty configuration, that is without any additional information about the data and the test sequences, AutoBlackTest revealed the highest number of faults together with the C_TpTnDpDn configuration. Both the Empty and C_TpTnDpDn configurations revealed the same two minor faults, but the Empty configuration revealed one severe and one minor fault that have not been revealed in the C_TpTnDpDn configuration, and the C_TpTnDpDn configuration revealed two minor faults that have not been revealed in the Empty configuration.

It is worth noticing that the faults revealed with AutoBlackTest in these experiments are real and not seeded faults, and are found in a popular application. Since the number of faults in jaolt is likely

Table VIII. Fault revealed with different configurations and initial test suites.

Configuration	Average faults	Total faults	Severe	Minor
Empty	2.33	4	1	3
C_Dp	1.33	2	0	2
C_DpDn	1.33	2	0	2
C_TpDp	2	2	0	2
C_TpDpDn	2	3	0	3
C_TpTnDp	0.67	2	0	2
C_TpTnDpDn	1.67	4	0	4

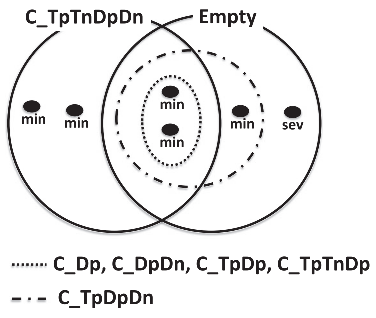


Figure 7. The relation among the faults discovered in the different configurations.

to be limited, and their presence could be concentrated in few specific areas of the application, revealing few more faults could be incidental rather than a sign of effectiveness. To disambiguate the case it was analyzed more in detail the state exploration in the Empty and C_TpTnDpDn configurations to check if there are intrinsic differences in the way the two approaches explore the execution space.

The diagram in Figure 8 indicates the distribution of the states visited in each configuration: the x-axis indicates the states incrementally numbered while visited, while the y-axis corresponds to the different configurations: Empty, C_TPDp (the configuration with the highest coverage) and C_TpTnDpDn (the configuration that revealed the highest number of faults). The diameter of each circle is proportional to the number of times the state has been visited.

The shape of the plotting shows an intrinsic difference between the exploration strategies with and without an the initial test suite. With an initial test suite, AutoBlackTest explores less states than without the initial test suite and covers each state more times with the exception of the initial states that are covered more in the absence of an initial test suite. With no initial test suite, AutoBlackTest shall autonomously understand how to use the application and thus spends more effort in the initial states. Then the exploration strategy proceeds in an unbiased way leading to the continuous discovery of new states. On the contrary with an initial test suite, AutoBlackTest spends more effort in the states close to the ones covered by the initial test suite and devotes less effort in discovering new states. Depending on the distance between the states covered by the initial test suite and the faulty states, one approach could be more likely to discover some faults than the other.

This phenomenon explains also why in the Empty configuration, AutoBlackTest discovered one fault not discovered by the other configurations. The fault uniquely discovered by the Empty configuration is triggered by interacting with the early states of the application, which are the states that the Empty configuration covers better, namely the states with a number less than 100 in Figure 8.

6.6. *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, third-party publicly available applications were selected. They have various sizes, cover different domains and have been already used in empirical studies on GUI testing.

Another threat to the internal validity is related to the choice of the values of the parameters that influence the behaviour of AutoBlackTest. To reduce this threat, the parameter values were chosen according to the empirical experiences reported in previous research (for the discount factor) and the empirical analyses reported in this paper (for the policy). Both previous research and the experiments reported in this paper show that the performance of Q-learning changes gracefully for small changes

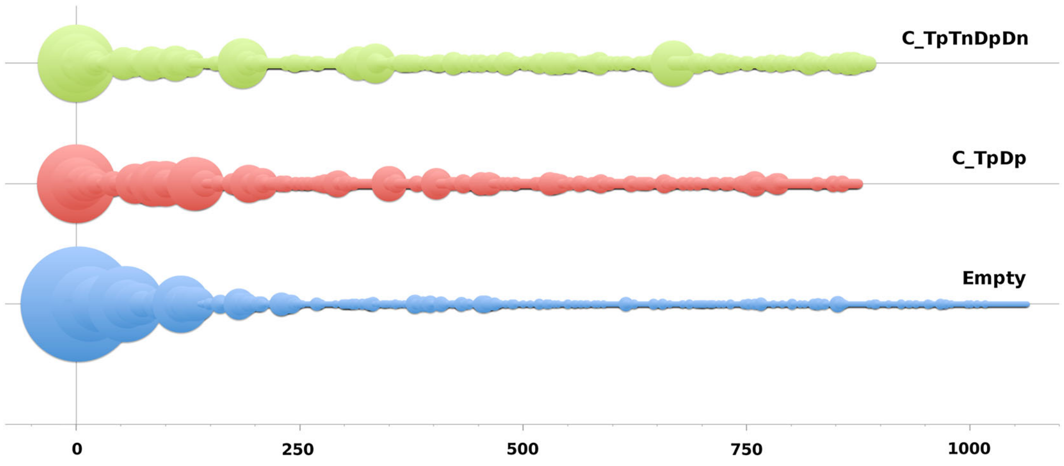


Figure 8. The distribution of the states visited in different configurations.

of these parameters, and thus, the impact of a choice on the results is limited. According to these evidences, although a strong statistical assessment was not performed, the results are expected to be stable with respect to small changes of the configurations.

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

Another threat to validity is related to the manual definition of the initial test suites. Since no system test suite was available for the case study, positive and negative test cases were manually defined according to our understanding of the application. The simplicity of the applicative domain, which refers to eBay auctions, mitigates the risk of misunderstanding the application. Naturally, test cases developed by a testing team would not suffer from this drawback since testing team members would be familiar with the application and would have access to the documentation.

The main threat to the external validity comes from the limited number of case studies that we considered. We experimented AutoBlackTest with five applications. 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 simulating overnight test sessions of 12 h and cannot be generalized to sessions of different length. In particular, it cannot be generalized to longer sessions.

Finally, the main threat to the construction validity is related to the possible presence of faults in the implementation of AutoBlackTest [23]. To mitigate this threat, the correctness of the behavioural model extracted for a number of sample executions was manually checked and the detected faults were validated by replaying all the executions that detected issues in the case studies.

7. RELATED WORK

Generating and executing system test cases for interacting applications is still largely a manual activity and automation is mostly limited to capture and replay tools, such as IBM Functional Tester [15] and Maveryx [24], that reexecute previously recorded test cases. 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 case generation exploits GUI coverage metrics that measure how many of the events produced by widgets have been stimulated in the testing process [4, 5]. 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, standard statement coverage was used to measure the amount of code executed by AutoBlackTest.

Memon *et al.* [6, 7] take advantage from GUI coverage metrics to define techniques that generate system test cases that cover sets of GUI events. 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 behaviour of the application.

A few recent approaches addressed the issue of generating GUI test cases applying search-based [8] and concolic testing techniques [9]. Although these techniques can derive test cases that directly improve code coverage, they have to face the issue of analyzing multiple layers of software in order to produce effective test cases. AutoBlackTest is a complementary solution that does not rely on the analysis of the code but generates test cases with a purely black-box perspective.

Xie and Memon investigated the use of oracles for GUI test cases. Xie *et al.* [20] show that there exists 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 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 [25]. 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 [26]. Other techniques produce test cases by relying on navigation models [27] or data captured from users sessions [28]. While these models and data are quite common for Web applications, they are less frequently available for GUI applications.

There is an impressive amount of work about the relation between learning of finite state models and testing [29–31]. So far the integration of learning and testing has been mostly exploited to generate tests in domains different than GUI testing and on cases smaller than entire software systems [32–36]. AutoBlackTest adds evidence of the effectiveness of combining learning and testing, also when applied to the generation of system test cases.

8. CONCLUSIONS

The problem of generating system test cases is particularly hard because black-box approaches have to deal with a complete GUI, which could be large and complex, and white-box approaches have to deal with the many software layers that are activated every time a stimulus is provided.

This paper presented AutoBlackTest, a black-box technique for automatically generating system test cases. AutoBlackTest effectively addresses the complexity and size of GUIs by incrementally learning how to interact with the application under test and thus incrementally synthesizing test cases that cover new functionalities of the applications. When a manually designed test suite is available, AutoBlackTest can take advantage of the test cases to synthesize complementary test cases.

The results presented in the 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. The empirically comparison with GUITAR indicates that AutoBlackTest can generate test cases that cover more statements and reveal more failures than the test cases generated with GUITAR.

Future research agenda include plans to extend AutoBlackTest to add the capability of automatically generating input values rather than using values present in a data set and to investigate the application of AutoBlackTest to the domain of apps for mobile devices.

REFERENCES

1. Tillmann N, Halleux JD. Pex: white box test generation for .NET. *Proceedings of the 2nd International Conference on Tests and Proofs (TAP),* Prato, Italy, 2008; 134–153.
2. Broy M, Jonsson B, Katoen JP, Leucker M, Pretschner A (eds.) *Model-based testing of reactive systems*, LNCS, vol. 3472. Springer Verlag: Berlin, Germany, 2005.
3. Taneja K, Xie T. DiffGen: automated regression unit-test generation. *Proceedings of the 23rd International Conference on Automated Software Engineering (ASE),* L'Aquila, Italy, 2008; 407–410.
4. Memon AM, Soffa ML, Pollack ME. Coverage criteria for GUI testing. *Proceedings of the 8th European Software Engineering Conference held Jointly with 9th ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC-FSE),* Vienna, Austria, 2001; 256–267.
5. Yuan X, Cohen MB, Memon AM. GUI interaction testing: incorporating event context. *IEEE Transactions on Software Engineering (TSE)* 2011; **37**(4):559–574.
6. Memon AM, Xie Q. Studying the fault-detection effectiveness of GUI test cases for rapidly evolving software. *IEEE Transactions on Software Engineering (TSE)* 2005; **31**(10):884–896.
7. Yuan X, Memon AM. Generating event sequence-based test cases using GUI run-time state feedback. *IEEE Transactions on Software Engineering (TSE)* 2010; **36**(1):81–95.
8. Gross F, Fraser G, Zeller A. Search-based system testing: high coverage, no false alarms. *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA),* Minneapolis, MN, USA, 2012; 67–77.
9. Anand S, Naik M, Yang H, Harrold MJ. Automated concolic testing of smartphone apps. *Proceedings of the International Symposium on Foundations of Software Engineering (FSE),* Cary, North Carolina, 2012; 1–11.
10. Sutton RS, Barto AG. *Reinforcement Learning: An Introduction*. MIT Press: Cambridge, MA, 1998.
11. Mariani L, Pezzè M, Riganelli O, Santoro M. AutoBlackTest: automatic black-box testing of interactive applications. *Proceedings of the Fifth International Conference on Software Testing, Verification and Validation (ICST),* Montreal, Canada, 2012; 81–90.
12. Watkins CJCH. Learning from delayed rewards. *Ph.D. Thesis*, King's College, Cambridge, UK, 1989.
13. Watkins CJCH, Dayan P. Technical note Q-learning. *Machine Learning* 1992; **8**:279–292.

14. Kephart JO, Chess DM. The vision of autonomic computing. *IEEE Computer* 2003; **43**(1):41–50.
15. IBM. IBM rational functional tester. Available from: http://www-01.ibm.com/software/awdtools/tester/functional/ [last accessed 2012].
16. Abul O, Polat F, Alhajj R. Multiagent reinforcement learning using function approximation. *IEEE Transactions on Systems, Man, and Cybernetics, Part C: Applications and Reviews* 2000; **30**(4):485 –497.
17. Lin LJ. Reinforcement learning for robots using neural networks. *Ph.D. Thesis*, Carnegie Mellon University, Pittsburgh, PA, USA, 1992.
18. Becce G, Mariani L, Riganelli O, Santoro M. Extracting widget descriptions from GUIs. *Proceedings of the International Conference on Fundamental Approaches to Software Engineering (FASE),* Tallinn, Estonia, 2012; 347–361.
19. Rothermel G, Untch RH, Chu C, Harrold MJ. Prioritizing test cases for regression testing. *IEEE Transactions on Software Engineering (TSE)* 2001; **27**(10):929–948.
20. Xie Q, Memon AM. Designing and comparing automated test oracles for GUI-based software applications. *IEEE Transactions on Software Engineering (TSE)* 2007; **16**(1):1–36.
21. Teachingbox. Available from: http://sourceforge.net/projects/teachingbox/ [last accessed 2012].
22. Xie Q. Developing cost-effective model-based techniques for GUI testing. *PhD Thesis*, University of Maryland, 2006.
23. Mariani L, Pezzè M, Riganelli O, Santoro M. AutoBlackTest: a tool for automatic black-box testing. *Proceedings of the International Conference on Software Engineering (ICSE) - Tool Demo,* Waikiki, Honolulu, HI, USA, 2011; 1013–1015.
24. Maveryx. Available from: http://www.maveryx.com [last accessed 2012].
25. Brooks AP, Memon AM. Automated GUI testing guided by usage profiles. *Proceedings of the International Conference on Automated Software Engineering (ASE),* Atlanta, Georgia, USA, 2007; 333–342.
26. Marchetto A, Tonella P, Ricca F. State-based testing of Ajax web applications. *Proceedings of the International Conference on Software Testing, Verification, and Validation (ICST),* Lillehammer, Norway, 2008; 121–130.
27. Andrews A, Offutt J, Alexander R. Testing Web applications by modeling with FSMs. *Software and System Modeling* 2005; **4**(3):326–345.
28. Elbaum S, Karre S, Rothermel G. Improving Web application testing with user session data. *Proceedings of the International Conference on Software Engineering (ICSE),* Portland, Oregon, 2003; 49–59.
29. Dallmeier V, Knopp N, Mallon C, Fraser G, Hack S, Zeller A. Automatically generating test cases for specification mining. *IEEE Transacitons on Software Engineering (TSE)* 2012; **38**(2):243–257.
30. Lorenzoli D, Mariani L, Pezzé M. Automatic generation of software behavioral models. *Proceedings of the International Conference on Software Engineering (ICSE),* Leipzig, Germany, 2008; 501–510.
31. Lo D, Mariani L, Santoro M. Learning extended FSA from software: an empirical assessment. *Journal of Systems and Software (JSS)* 2012; **85**(9):2063 –2076.
32. Groz R, Irfan MN, Oriat C. Algorithmic improvements on regular inference of software models and perspectives for security testing. *Proceedings of the International Symposium on Leveraging Applications of Formal Methods, Verification and Validation (ISoLA),* Heraklion, Crete, Greece, 2012; 444–457.
33. Meinke K, Sindhu M. LBTest: a learning-based testing tool for reactive systems. *Proceedings of the International Conference on Software Testing, Verification and Validation (ICST),* Luxembourg, 2013; 447–454.
34. Meinke K, Niu F, Sindhu M. Learning-based software testing: a tutorial. *Proceedings of the International Symposium on Leveraging Applications of Formal Methods, Verification and Validation (ISoLA),* Vienna, Austria, 2011; 200–219.
35. Walkinshaw N, Bogdanov K, Derrick J, Paris J. Increasing functional coverage by inductive testing: a case study. *Proceedings of the International Conference on Testing Software and Systems (ICTSS),* Natal, Brazil, 2010; 126–141.
36. Mariani L, Papagiannakis S, Pezzé M. Compatibility and regression testing of COTS-component-based software. *Proceedings of the International Conference on Software Engineering (ICSE),* Minneapolis, Minnesota, 2007; 85–95.