2015 30th IEEE/ACM International Conference on Automated Software Engineering

Dynamically Testing GUIs Using Ant Colony Optimization

Santo Carino[*] and James H. Andrews[*†]

[*]Department of Computer Science
University of Western Ontario
London, Ontario, Canada
Email: scarino,andrews@csd.uwo.ca

[†]Google Inc.
1600 Amphitheatre Parkway
Mountain View, CA, USA

***Abstract*—In this paper we introduce a dynamic GUI test generator that incorporates ant colony optimization. We created two ant systems for generating tests. Our first ant system implements the normal ant colony optimization algorithm in order to traverse the GUI and find good event sequences. Our second ant system, called AntQ, implements the antq algorithm that incorporates Q-Learning, which is a behavioral reinforcement learning technique. Both systems use the same fitness function in order to determine good paths through the GUI. Our fitness function looks at the amount of change in the GUI state that each event causes. Events that have a larger impact on the GUI state will be favored in future tests. We compared our two ant systems to random selection. We ran experiments on six subject applications and report on the code coverage and fault finding abilities of all three algorithms.**

I. Introduction

Software testing is a large and expensive part of the development life-cycle [28]. A lot of research has gone into traditional black- and white-box testing. Such methods include using dynamic symbolic execution [32] [11] [37], using mutants [21] [15] [2], random testing [4] [3] [16], and search-based software testing [14] [24]. With graphical user interfaces (GUI) being commonplace in most modern software, ranging from the web, to mobile, and desktop, the need for GUI tests has arisen, which further increases the cost. As with automated black-box test generation, there is a need for automated test generators for GUIs. While traditional test generators are able to generate tests similar to tests a programmer would write, that is a series of methods calls and variable declarations, GUI tests typically consist of a series of events to be run on the system, for example, clicking buttons or entering data into text boxes. Currently, GUI tests can be written manually or created using a record and playback system [31] [1]. Manually written tests consist of methods to locate a specific GUI widget and executing its event. Record and playback systems record the user's interaction with the application and can then play back the test script at a later time. Both methods are time-consuming.

In this paper we introduce a test sequence generator for GUIs. The system uses ant colony optimization [13] in order to generate tests that have an impact on the state of the GUI. Ant systems have been shown to be useful for finding paths through a graph. Since GUIs are best represented by graphs, where each vertex is an event and each edge an action, it follows that ant colony optimization may be a good method to explore the GUI graph. Furthermore, we can represent the GUI test generation problem as an optimization problem, such as trying to generate tests that maximize code coverage. Our system's goal is to find sequences of events that have a large impact on the GUI state.

There are generally two types of architectures for testing GUI systems: model-based systems and dynamic systems [5]. Model-based systems will first try to discover the structure of the GUI hierarchy and generate a model. The models are generally represented as event-flow graphs or state machines. Test cases are then generated offline based on the model's structure. Model-based systems can generate infeasible test cases due to the fact that some events can only be selected if the system is in a specific state. A dynamic testing system will explore the GUI as it generates tests and create a model as events are discovered. Since the system can determine which events are available at each state, the tests are always feasible. Due to the benefits of the dynamic architecture, we opted to use it, and therefore all tests generated in our system are feasible.

The contributions of this paper are as follows.

- We adapt the concept of using the GUI state change ratio to ant colony systems and show how it can be used as an effective fitness function.
- We apply two ant colony systems to GUI testing: the Ant System and AntQ, and show how they can generate effective test cases.
- We evaluate both ant systems and compare them against random search.

The paper is organized as follows. Section II discusses related work and other GUI testing systems. Section III shows how we calculate our state change values, which we use as a fitness function. Section IV looks at the two ant systems we implemented: a traditional Ant System and AntQ, which combines ant colony optimization and Q-Learning. Section V is where we evaluate our system by testing the two algorithms on six subject applications and comparing their results against one another, as well we compare against random selection. Finally, we conclude in Section VI.

II. Background and Related Work

Here we describe ant colony optimization in general, as well as other systems designed for automated GUI test sequence generation.

A. Ant Colony Optimization

An ant colony optmization algorithm [13] is a swarm intelligence system typically used to solve problems that can be represented as a graph, such as the travelling salesman problem. The ant system's main components are positive feedback and distributed computation. Furthermore, the ant system makes use of a greedy heuristic when traversing a graph. The system uses a set of agents, called ants, to explore the graph in order to find an optimal path. Once a generation of ants has completed its search, the edges of the graph are updated. The edges are updated by having a value, called pheromones, deposited on them. The amount of pheromone deposited on an edge depends on the fitness of the ants that traversed it. The fitness function used for many problems is the inverse of the tour length for each ant. The more ants that traverse an edge, the more pheromone it will receive. Furthermore, each edge's pheromones reduce over time due to evaporation. Another component of the ant system is the greedy heuristic. The heuristic is used during the selection process in order to determine the next vertex to visit. The heuristic value is combined with the pheromone value, which determines the vertex's desirability. The heuristic is generally the inverse of the edge's distance, so that shorter edges will be favored.

Ant systems have been combined with Q-Learning [12] to further improve their results. Q-Learning [34] is a behavioral reinforcement algorithm that teaches agents how to act in controlled Markovian domains. It works by continuously improving evaluations of specific actions in specific states. Our ant system incorporates Q-Learning into each ant's exploration. In the traditional ant system, the pheromone values are only updated after each ant has successfully completed its run. In antq, the edges are updated at two points. First, when an ant traverses an edge it immediately updates the edge's pheromone using the Q-Learning technique. Second, once each ant has completed its run, the edges are updated again as in the classic ant system. Another difference between the two systems is that in the ant system, all edges are updated after each generation, whereas in antq only edges that were traversed by an ant are updated.

B. GUI Testing Methods

A popular and well researched model-based testing system is GUITAR [29]. GUITAR works by ripping [26] the GUI into a GUI hierarchy file, which can then be translated to an event-flow graph (EFG)[25]. EFGs represent all possible event interactions that can occur. The graph can be traversed and tests generated that consist of a list of events to execute. Various techniques can be used to generate tests, such as using covering arrays [38] or using automated planning to cover defined goals [27]. The major flaw with GUITAR is that it can generate infeasible tests. These broken tests can be repaired [20]; however, the process can be expensive.

Bauersfeld et al. [6] developed a system that incorporates ant colony optimization in order to test GUIs. The goal of their system is to generate test sequences that result in a large maximum call tree (MCT). MCTs are trees that represent method calls in a system. A large MCT is a tree with many leaf nodes. The idea behind generating large MCTs is that methods can be called in different contexts, such as from other methods, and by generating large MCTs the context in which a method is called is likely to change. For example, a method $m1()$ can be called by both methods $m2()$ and $m3()$. It would be beneficial to have $m1()$ called in both contexts to ensure it is properly tested as either one of the calling methods can change variables or objects that $m1()$ relies on. The authors were successful in generating test sequences that generated large MCTs.

Gross et al. [17] developed EXSYST, which is a dynamic test sequence generator that uses a genetic algorithm to evolve an entire test suite. The authors begin by showing that using a unit test generator such as RANDOOP [30] can result in many false failures, as the implicit contraints of the AUT imposed by the GUI are ignored. EXSYST works by generating random test suites and determining their fitness based on branch distance. However, the system takes into account multiple target branches instead of just a single branch. The system performs genetic manipulation in the form of crossover and mutation. Crossover happens at the test suite level by having test suites exchange a number of test cases. Furthermore, both test suites and test cases can be mutated. Test suites are mutated by adding test cases or modifing an existing test case. Test cases are mutated by deleting, changing, or inserting operations. The mutation phase can cause tests to break. In order to fix infeasible test cases, a model of the GUI is stored. The model contains sequences of actions and states that those actions lead to. The model can be used to repair broken test cases.

Mariani et al. [22] developed AutoBlackTest, which is built on top of IBM Rational Tester, to automate GUI test sequence generation. AutoBlackTest uses Q-Learning to find good sequences of events. The system works by exploring the GUI and assigning values to edges based on a reward and a Q function. The reward is the amount of change that occurs in the GUI when an event is executed. The Q-value is assigned based on the event's reward and the Q-value of its best successor event. The system is also able to start testing from any found state. As a result, AutoBlackTest stores the states found and the steps required to enter that state. When a new test is being generated, it will attempt to start the test at a previously found state; however, if this is not possible, it starts testing from its current state. The authors incorporate heuristic actions for different event types, where the action can encompass many different events (for example, for handling file open and save dialogs). The authors compared AutoBlackTest against GUITAR and show that it is able to achieve higher levels of code coverage and find more faults.

III. GUI State Change as a Fitness

The goal of ant colony optimization algorithms is to try to find an optimal path through a graph, for example the shortest path to solve the travelling salesman problem. In GUI testing systems, the value to optimize can differ. In the case of [6], the authors try to maximize Maximum Call Trees. In other meta-heuristic search systems, such as EXSYST [17], the authors try to maximize code coverage. Our system attempts to maximize the amount of change in the GUI state for each test case. That is, it will prefer interacting with widgets that have a large impact on the GUI, such as events that enable new widgets or open windows. This concept is borrowed directly from AutoBlackTest [22].

We can measure the amount of change on a system's GUI by inspecting the GUI's state before an event is run, running the event, and inspecting the resulting GUI state. The state change value is then calculated as the difference between the two states. The system takes into account two factors when calculating the state change value: first, it compares the properties of the widgets that exist in both states; second, it looks for any new widgets that exist in the new state.

A GUI's state can be broken down into its individual widgets, such as buttons, sliders, and panels. Each widget contains a set of properties that describe it. For example, a button's properties will contain its text, whether or not it is enabled, its position, and so on. Furthermore, some properties, which we call *traits*, can be used to identify a widget. This is needed because when we compare states, we want to ensure we are comparing the same widgets. Formally we say a state S contains a set of widgets $<w_1, w_2, ..., w_n>$, where each widget w contains a set of properties $<p_1, p_2, ..., p_n>$, we say that a property p is a trait if it can be used to identify the widget.

When comparing two widgets, w_1 and w_2, we say $w_1 =_t w_2$ iff $\mid w_1 \mid = \mid w_2 \mid \wedge \forall t_i \in w_1, \exists t_j \in w_2$ s.t. $t_i = t_j$, where t is a trait in w. That is, two widgets are equivalent if all of their traits are equal. If one widget has a different number of traits than the other, the two widgets are considered to be different.

When comparing two states, we only look at a subset of the widgets and their properties, which we call an *Abstract State*. An abstract state, AS, contains a set of of *Abstract Widgets*, where each abstract widget contains a subset of a real widget's properties. We only use a subset of a widget's properties for state comparison, as not all properties are relevant. For example, an $AbstractTextBox$ will contain the text box's text, GUI hierarchy position, class, and whether or not it is enabled and editable.

If we have two abstract states, $AS = \{w_1, ...w_n\}$ and $AS' = \{w'_1, ...w'_n\}$, we define the restriction operator $AS \backslash_t AS' = \{w_i \mid w_i \in AS \wedge \nexists w_k \in AS'$ s.t. $w_i =_t w_k\}$. This restriction operator is used in our calculation of state change ratios.

Equation (1) is used to calculate the ratio of change between two widgets if $w_1 =_t w_2$. The equation determines the ratio of change based on each widget's properties.

$$\text{diff}_w(w_1, w_2) = \frac{\mid P_1 \backslash P_2 \mid + \mid P_2 \backslash P_1 \mid}{\mid P_1 \mid + \mid P_2 \mid} \tag{1}$$

where P_i is a property in w_i.

We use (2) to compare two states and determine the ratio of change. The equation takes into account the ratio of change between widgets that appear in both states, as well as new widgets that appear in the new state. It does not include widgets that may have been removed from the first state, such as when a window is closed. This is done to favor widgets that allow access to more events.

$$\text{diff}_{AS}(AS_1, AS_2) = \frac{\mid AS_2 \backslash_t AS_1 \mid + \sum_{w_1 \in AS_1, w_2 \in AS_2, w_1 =_t w_2} \text{diff}_w(w_1, w_2)}{\mid AS_2 \mid} \tag{2}$$

The final change ratio is used as the fitness to our ant system. The benefit of using the state change ratio as a fitness is that it only relies on the information within the GUI and it does not require any code instrumentation.

IV. Ant Systems

A. Architecture

The architecture of our system is shown in Fig. 1. Here we describe each component in detail.

Application Under Test

The application under test (AUT) is the program being tested. In our case it is a Java Swing application, but it can be generalized to any GUI system.

Window Listener

The Window Listener implements the *WindowListener* interface provided with the JDK. It is able to listen for window events such as when a window opens, closes, or is activated. Every time a window update event happens, the window listener captures the affected window and scrapes it. The window listener determines if the window is modal or not by checking the window's properties. A modal window is a window that has control over the application. No events outside of that window are able to be executed until the window has closed. An example modal window is the the save dialog window used in many applications. If a window is modal, only its events will be scraped, otherwise all windows are scraped.

GUI Scraper

The GUI Scraper accepts inputs from the Window Listener. The Window Listener passes the scraper the set of windows to be scraped. For each window, the scraper extracts the window's widgets and determines if the widget is executable, such as a button, or if it is a container, such as a panel. Widgets that are executable are stored in a widget list, whereas containers are scraped further. A window can contain many layers of containers. The scraper recursively scrapes each container until it reaches the bottom of the GUI hierarchy.

The scraper can work in two modes: full state extraction or executable widgets only. The 'full state' mode will extract all available widgets and containers and return them. This data is used to calculate the state change ratio for the ant algorithms. The second mode, 'executable widgets only', extracts and returns widgets that can be executed and are valid. By valid we mean enabled and visible.

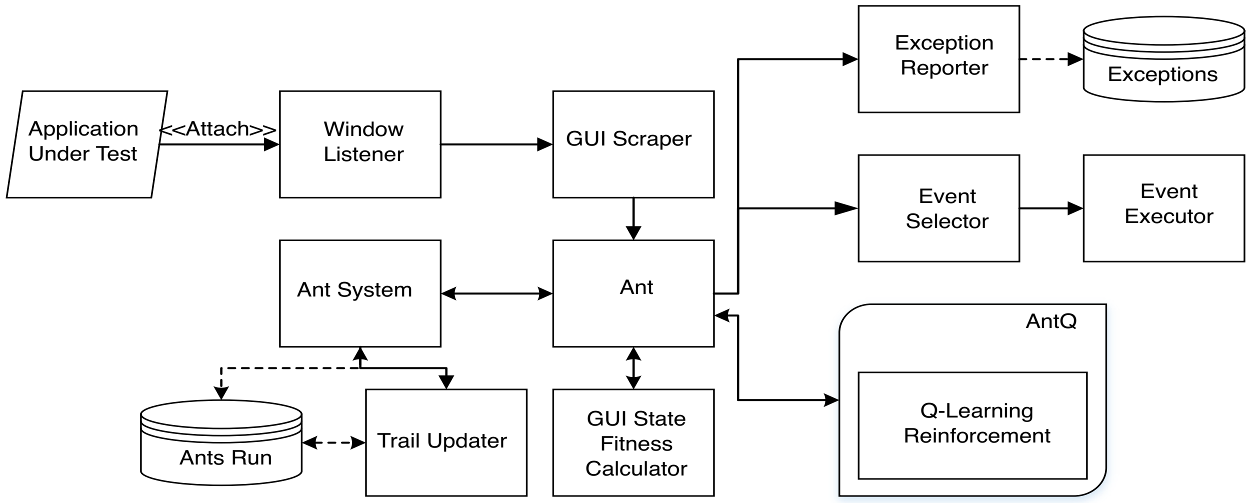

Fig. 1: Ant System Architecture.

Ant System

The Ant System is the controller for the set of ants that will test the AUT. The Ant System deploys each ant and sets the ants tour length based on the configuration set by the test engineer. When an ant has completed its tour, the edges traversed by the ant and the cumulative state change ratio are stored in an ant database called 'Ants Run'. After the generation of ants has completed its run, the edges of the graph are updated using the Trail Updater and the ant database.

Trail Updater

The Trail Updater is the component responsible for updating the edges traversed by the set of ants in the ant database. The updater only uses the ants from the current generation to update the trails, and not the ants from any previous generation. The Trail Updater looks at the edges traversed by each ant and deposits pheromones. If an edge was not traversed by an ant, its pheromone is reduced. The amount of pheromone deposited depends on the algorithm being used (Ant System or AntQ), and the pheromone evaporation rate.

Ant

The Ant represents the actual test case. After being spawned by the Ant System, the Ant receives input from the scraper about the available executable events. Before executing an event, the Ant generates an abstract state by retrieving the AUT's state data from the scraper module and passing it to the GUI State Fitness Calculator. The Ant then passes the available events to the Event Executor. After execution is complete, the Ant captures the resulting state and retrieves the abstract state from the fitness calculator. The two abstract states are passed to the calculator module in order to determine the ratio of change. The resulting difference is added to the cumulative state change of the test case, which is to be used by the trail updater. The ant continues this process until it reaches its tour length.

GUI State Fitness Calculator

The GUI State Fitness Calculator takes as input the raw widget data provided by the GUI Scraper module. The calculator extracts the necessary properties and creates an abtract state representation of the state and returns it to the Ant module. The calculator can also compare two abstract states and return the difference. The return value is in the range [0,1].

Event Selector

The Event Selector takes as input the list of available events that can be executed. It applies the pseudo random proportional rule to the list of events and selects an event to execute. The selected event is sent to the Event Executor.

Event Executor

The Event Executor takes as input an event to execute. The event is compared against a list of possible event types, such as buttons, sliders, and text boxes. Depending on the type of the event, a different action is taken. Some events can have multiple actions, for example, a table can have a single cell filled, an entire row or column filled, or the entire table can be filled with data. For complex events we use a random selection in order to determine the type of action to take. In the case of the table, each action would have a 25% chance of being selected. The data used to fill in text boxes and tables is selected at random from a predefined list of inputs. Table I shows the list of events and their respective actions.

The Event Executor has heuristics in place to deal with two specific scenarios: file dialogs and color choosers. When the system detects either type of window, it executes a set of defined actions. For file dialogs, the executor sets the directory to 'home', types in a random file name, selects a random file type, and clicks the confirm (save/open) button. For the color chooser window, the executor simply clicks the 'ok' button. The first heuristic is in place so that the test does not pollute

TABLE I
Event Actions

Event	Action
Button, Spinner, Toggle Button	Click()
Text Area, Password Field	Type(input)
Table	Type(cell/row/column/table)
List	Select(row/rows)
Tab Panel	SelectTab(tabIndex)
Slider	SetSelection(value)
Combo Box	SetSelection(index)

the host environment. Without the heuristic it is possible that important files or directories will be overwritten or renamed, which would interfere with the testing. The color chooser heuristic is used to avoid waisting time in the dialog window, since the color chooser has little affect on the test, if any.

Q-Learning Reinforcement

The Q-Learning Reinforcement module is called only by the AntQ algorithm. During the run of the AntQ algorithm, when an event is executed, the edge traversed is immediately updated. The module updates the specific edge and returns to the Ant module.

Exception Reporter

The Exception Reporter implements the *UncaughtExceptionHandler* interface provided with the JDK. Any exception thrown during the execution of an event is caught. The reporter logs the exception message, stack trace, and the list of events that lead up to the event being thrown. The data is then stored in a database.

B. Event Selection

Events are selected by using the *pseudo random proportional rule* shown in (3). We use the pseudo random proportional rule as it has been shown to be more effective than using the random proportional rule alone [12].

$$y = \begin{cases} \max\limits_{y \in allowed_x} \{\tau_{xy}^{\alpha} \cdot \eta_{xy}^{\beta}\} & \text{if } q \leq q_0 \\ Y & \text{otherwise} \end{cases} \tag{3}$$

where $allowed_x$ is the set of events that can follow event x, τ is the pheromone value, η is the heuristic value, q is a random value where $0 \leq q \leq 1$, and q_0 is a value determined by the engineer beforehand. This rule states that we select the best edge with probability q, or we select an edge using the *random proportional rule* using probability $1-q$. The equation for the random proportional rule is (4). The rule states that an event is selected randomly based on its proportion determined by its pheromone and heuristic values.

$$P_{xy} = \begin{cases} \frac{\tau_{xy}^{\alpha} \cdot \eta_{xy}^{\beta}}{\sum_{u \in allowed_x} (\tau_{xu}^{\alpha} \cdot \eta_{xu}^{\beta})} & \text{if } y \in allowed_x \\ 0 & \text{otherwise} \end{cases} \tag{4}$$

Our system does not implement a hueristic, so only the edge's pheromone values are taken into account when selecting events. We used a default pheromone value of 0.5 for every edge in the graph. Over time the value will increase or decrease depending on the fitness function.

C. Updating Pheromones

The main difference between the Ant System and the AntQ system is in how they update their pheromone values. The Ant System updates its pheromones using (5).

$$AQ(x,y) = (1-\alpha) \cdot AQ(x,y) + \Delta AQ(x,y) \tag{5}$$

where α is the pheromone evaporation rate.

When all ants of the current generation have finished their run, the system updates *all* of the edges in the graph found so far. The edges are updated using (6), which states that edges appearing in least one run of the k best ants, where $k \leq n$ and where n is the number of ants in a generation, receives pheromones. The amount of pheromones received is equal to the average fitness of the ant runs that the edge appears in. For example, if an ant appears in three of the k_{best} ant runs, it will receive pheromone equal to the average fitness of those three runs. Edges not appearing in any of the k_{best} runs have their pheromones reduced based on the evaporation rate α.

$$\Delta AQ(x,y) = \begin{cases} \frac{\sum_{i=1}^{k} SC(i)}{k} & \text{if (x,y) cov. by any } ant \in k_{best} \\ 0 & \text{otherwise} \end{cases} \tag{6}$$

where k_{best} is the set of the best k ants in the current generation.

The AntQ system updates its pheromones using (7). As the ant traverses the graph, it updates each edge it touches. This is the reinforcement learning borrowed from Q-Learning. During this stage the $\Delta AQ(x,y)$ is 0 for all edges. Once all ants in the generation have completed their run, the system updates only those edges that were traversed by an ant, all other edges keep their current pheromone value. The traversed edges are updated using (7), where $\gamma \cdot \underset{u \in allowed_y}{\text{Max}} AQ(y,u)$ is 0 for all edges.

$$AQ(x,y) = (1-\alpha) \cdot AQ(x,y) + \alpha \cdot \left(\Delta AQ(x,y) + \gamma \cdot \underset{u \in allowed_y}{\text{Max}} AQ(y,u)\right) \tag{7}$$

where γ is the cooling factor, which is a part of the Q-Learning model.

For both the Ant System and AntQ system, the amount of pheromone deposited is calculated using (8).

$$SC(ant_i) = \frac{StateChange(ant_i)}{TourLength(ant_i)} \cdot UniqueEvents(ant_i) \tag{8}$$

```
function ANTCOLONY(Generations, Ants, Length)
    for $i \in Generations$ do
        $stateDiffs \leftarrow \{\}$
        for $j \in Ants$ do
            $cuDiff \leftarrow 0$
            $StartAUT()$
            for $k \in Length$ do
                $event \leftarrow selectEvent()$
                $curState \leftarrow getState()$
                $execute(event)$
                $newState \leftarrow getState()$
                $diff \leftarrow getDiff(curState, newState)$
                $cuDiff \leftarrow cuDiff + diff$
                $curState \leftarrow newState$
            end for
            $ShutdownAUT()$
            $stateDiffs.add(cuDiff)$
        end for
        $UpdateAllTrails(stateDiffs)$
        //Eq. 5. Ant Collaboration
    end for
end function
```

Fig. 2: Ant System Algorithm

where ant_i is the index of the i-th best ant run in the current generation, $StateChange()$ is the cumulative state change of the entire ant's run, $TourLength()$ is the length of the ant's run, and $UniqueEvents()$ is the number of unique events executed during the ant's run, where $0 \leq UniqueEvents() \leq TourLength()$. When calculating the state change using the $StateChange()$ method, the system only counts the amount of change once for each event in an ant's run, regardless if the event is executed multiple times. This is to discourage tests repeating the same, high state changing events. Furthermore, we further favor unique events by dividing by the length of the tour and multiplying by the number of unique events. This will favor tests with more unique events.

Algorithms 2 and 3 highlight the differences between the two ant systems. During the AntQ algorithm, the Q-Learning reinforcement is updated using (7) after each edge is traversed, where the $\Delta AQ(x, y)$ value is 0. At the end of each generation, the same equation is used to update the ant trails, however, the $\gamma \cdot \underset{u \in allowed_y}{\text{Max}} AQ(y, u)$ values are 0 for all edges.

V. EVALUATION

A. Applications Under Test

We evaluated both ant systems, as well as random selection, on six subject applications. The details of the applications can be seen in Table II. The line and branch information was found using Cobertura [10], which we used to measure the code coverage. Cobertura counts the executable lines of code and ignores the rest. The six applications chosen have all appeared in the literature previously. They are of moderate size and represent real-world applications.

B. Parameter Tuning

In order to determine which values to set for the available tuning parameters, we ran a small set of experiments on TerpSpreadsheet, TerpWord, and ArgoUML. We ran test suites of size 100 that consisted of tests of length 20. We used 10 ants per generation and we used the best 7 ants to update the edges. We repeated the process five times and took the average number of statements covered. The tuning parameters for the Ant System are the pheromone evaporation rate (α) and the pseudo random proportional selection value (q_0). Both values are in the range [0,1]. For the AntQ algorithm, an additional parameter is required, which is the cooling factor (γ). For all three parameters, we chose the values 0.3, 0.6, and 0.9. Since we do not know the effects of these parameters in this context, we chose a spread of values ranging from low to high. For the Ant System, we combined each α value against each q_0 value, which resulted in nine combinations. For the AntQ system we used a 2-way combinatorial system to find all pairs between the three parameters. This, too, resulted in nine combinations. We chose to use a 2-way combinatorial selection for the factors as the number of experiments is too large to fully test in a reasonable amount of time. It is possible that values or combinations of values not chosen would perform

```
function ANTQ(Generations, Ants, Length)
    for $i \in Generations$ do
        $stateDiffs \leftarrow \{\}$
        for $j \in Ants$ do
            $cuDiff \leftarrow 0$
            $StartAUT()$
            for $k \in Length$ do
                $event \leftarrow selectEvent()$
                $curState \leftarrow getState()$
                $execute(event)$
                $newState \leftarrow getState()$
                $diff \leftarrow getDiff(curState, newState)$
                $cuDiff \leftarrow cuDiff + diff$
                $curState \leftarrow newState$
                $updateTrail(k-1, k)$
                //Eq. 7. Q-Learning Reinforcement
                //$\Delta AQ(x, y)$ is 0
            end for
            $ShutdownAUT()$
            $stateDiffs.add(cuDiff)$
        end for
        $UpdateTouchedTrails(stateDiffs)$
        //Eq. 7. Ant Collaboration
        //$\gamma \cdot \underset{u \in allowed_y}{\text{Max}} AQ(y, u)$ is 0
    end for
end function
```

Fig. 3: AntQ Algorithm

TABLE II
Application Summary

Application	Classes	Statements	Branches
ArgoUML	1233	54632	24074
Buddi	1529	101949	43670
Gantt Project	689	27804	8926
TerpSpreadsheet	137	5449	2135
TerpWord	208	10340	3625
TimeSlotTracker	487	10090	3115
Total	4283	210264	85545

TABLE III
Ant System Parameter Tuning

		Statements		
α	q_0	**TerpS**	**TerpW**	**Argo**
0.3	0.6	3627	5750	19578
0.6	0.9	3592	5823	18982
0.9	0.3	3563	5715	19807
0.3	0.3	**3711**	**6286**	20011
0.6	0.3	3666	6107	**20264**
0.9	0.6	3508	5564	19096
0.3	0.9	3526	5893	19075
0.6	0.6	3598	5767	20041
0.9	0.9	3444	5511	18338

TABLE IV
AntQ Parameter Tuning

			Statements		
α	γ	q_0	**TerpS**	**TerpW**	**Argo**
0.3	0.9	0.6	3791	6423	20258
0.6	0.6	0.9	3792	6304	19293
0.9	0.3	0.3	3801	6230	19629
0.3	0.3	0.9	3801	6348	19395
0.6	0.3	0.6	3814	6416	19450
0.9	0.6	0.6	3765	6248	19466
0.6	0.9	0.3	3774	6412	19916
0.3	0.6	0.3	3788	6458	**20170**
0.9	0.9	0.9	**3833**	**6471**	20031

better; however, we are only looking for guidance and not for the optimal values for each application.

The results for the Ant System are shown in Table III. The parameter values that resulted in the most coverage are α = 0.3 and q_0 = 0.3 for both TerpSpreadsheet and TerpWord, and α = 0.6 and q_0 = 0.3 for ArgoUML. We performed a factorial ANOVA and Tukey test to determine if the factors had a significant impact on the results, as well as to determine if any value of each factor was significant. The results of the ANOVA showed that in all three applications, both the α and q_0 factors had a significant impact on the statements covered, but that their interactions did not. Parameter values for α of 0.3 and 0.6 and a q_0 of 0.3 were found to have a significant impact for TerpSpreadsheet. For TerpWord, an α value of 0.3 and a q_0 value of 0.3 were significant. Finally for ArgoUML, α values of 0.3 and 0.6 and a q_0 value of 0.3 had significant impacts on the coverage.

The results for the AntQ algorithm are shown in Table IV. The parameter values that resulted in the most statements covered were α = 0.9, γ = 0.9, and q_0 = 0.9 for both TerpSpreadsheet and TerpWord, and α = 0.3, γ = 0.6, and q_0 = 0.3 for ArgoUML.

Again we performed a factorial ANOVA and Tukey test. The ANOVA showed that no single factor had a significant impact on the results for all three applications. The Tukey test results for both TerpSpreadsheet and TerpWord showed that no value for each parameter resulted in significantly more coverage. The Tukey test results for ArgoUML showed that an α value of 0.3 and a γ value of 0.9 were significantly better.

TABLE V
Test Suite

Algorithm	α	γ	q_0	**Ants**	**k-best**	**Length**	**Tests**
Random	-	-	-	-	-	30	300
Ant System	0.3	-	0.3	20	15	30	300
AntQ	0.3	0.9	0.3	20	15	30	300

TABLE VI
Statement Coverage (LOC)

Application	**Random**	**Ant System**	**AntQ**
ArgoUML	20950	21209	**21610**
Buddi	15008	15185	**15565**
Gantt Project	17992	17330	**18028**
TerpSpreadsheet	**4010**	3900	3957
TerpWord	7076	6966	**7264**
TimeSlotTracker	6837	6638	**6910**
Total	71873	71228	**73334**

C. Random Testing

The random search algorithm in which the two ant algorithms are compared is built on top of the same dynamic system. The random module is given the available, valid components to execute, selects one at random, and passes it to the event executor. We chose to compare against random selection as the evaluation criteria for meta-heuristic search [19] state that any proposed algorithm should perform better than random search.

D. Test Suites

Table V shows the data regarding the test suites and parameters chosen. Each algorithm was run using tests of length 30 and each suite contained 300 test cases. For the two ant algorithms, we used 20 ants per generation and used the best 15 ants when updating the trail values. We repeated the process six times for each AUT to account for the random nature of each algorithm. We did not limit the running time, but rather allowed each algorithm to run the same number of events. This was done to ensure fairness.

E. Coverage Metrics

The statement coverage results are shown in Table VI. In five out of six applications, AntQ performed better than random selection. Random selection performed better than both ant algorithms for TerpSpreadsheet. We performed a Tukey test comparing each algorithm and found that AntQ was significantly better than random selection and the Ant System for both ArgoUML and Buddi. Random selection was not significantly better than AntQ for any AUT; however, it did perform significantly better than the Ant System for both TerpSpreadsheet and TimeSlotTracker.

In five out of the six applications under test, AntQ covers more statements than random selection.

Figs. 4 to 9 show the increase in coverage over time for all three algorithms. In the case of ArgoUML, Buddi, Gantt

TABLE VII
Statement Coverage Comparison (%)

AUT	Rand	Ant	AntQ	GUITAR	EXSYST	ABTest
Argo.	38	39	**40**	25	-	-
TerpS.	**74**	72	73	28	39	-
TerpW.	68	67	**70**	27	54	-
TimeS.	**68**	66	**68**	55	-	**68**

TABLE VIII
Exceptions Found - Average(Total)

Application	Random	Ant System	AntQ
ArgoUML	8.8(19)	9.8(18)	10.7(17)
Buddi	0	0	0
Gantt Project	0	0	0
TerpSpreadsheet	6(10)	5.8(8)	6(11)
TerpWord	9.7(13)	11.5(15)	10.7(15)
TimeSlotTracker	4.2(7)	3.5(8)	5.3(9)
Average(Total)	4.8(49)	5.1(49)	5.4(52)

Project, TerpWord, and TimeSlotTracker, the AntQ system is consistently better than random selection. In the case of TerpSpreadsheet, the AntQ and random algorithms grow at a similar pace.

In five out of the six applications under test, AntQ's coverage growth dominates random selection's coverage growth.

We compare our statement coverage results with those published in [18] [22] [29]. The results can be seen in Table VII. In the case of GUITAR and EXSYST, our three algorithms outperform them in all cases. In the case of AutoBlackTest, our system finds the same level of coverage.

F. Unhandled Exceptions

Table VIII shows the number of uncaught exceptions found. Neither Buddi nor Gantt Project threw any uncaught exceptions.

The ant algorithms found on average an equal or greater number of exceptions compared to random selection.

AntQ found a greater number of uncaught exceptions than both the Ant System and random selection.

Again we performed a Tukey test and found that AntQ found a significantly higher number of exceptions than the Ant System for TimeSlotTracker. No other pairs of algorithms for any AUT were found to have significant differences.

G. Cost

Table IX shows the average running time for each test suite on each application. The random selection algorithm outperforms both the Ant System and AntQ algorithms in all cases. The results are not surprising as both ant algorithms must maintain a trail graph. Furthermore, the ant algorithms have more complex selection procedures.

TABLE IX
Average Time to Run (Hours)

Application	Random	Ant System	AntQ
ArgoUML	5.5	5.6	5.6
Buddi	4	5.4	5.2
GanttProject	4	4.5	4.7
TerpSpreadsheet	3.5	3.9	3.9
TerpWord	3.4	4.1	3.7
TimeSlotTracker	3.9	4.6	4.6
Total	24.3	28.1	27.7

TABLE X
Fitness Function Evaluation

Application	Pearson Value
ArgoUML	0.414
Buddi	-0.026
Gantt Project	0.339
TerpSpreadsheet	-0.213
TerpWord	0.228
TimeSlotTracker	0.497

H. Evaluating the Fitness Function

In order to determine if our fitness function is effective, we measured the correlation of the cumulative state change ratio with the statement coverage for every test case for each application using AntQ. We compared the values using the Pearson test with $\alpha = 0.05$. Table X shows the results.

In the cases of ArgoUML, Gantt Project, TerpWord and TimeSlotTracker, there is a weak to strong positive correlation between the state change ratio and the statement coverage. In the case of Buddi and TerpSpreadsheet, there is a weak negative correlation between the two values. Figs. 10 to 15 show the scatter plots comparing the state change ratio values to the number of statements covered for the AntQ algorithm. There is evidence that the fitness function provides positive results; however, the effectiveness of the function depends on the application under test.

I. The Effectiveness of Random Testing

This evaluation shows that random selection for dynamic GUI testing can be effective in finding faults. Little to no research has been conducted on the viability of random selection for GUIs. Many of the systems discussed in the literature do not compare against random selection. AutoBlackTest [22] is compared against GUITAR; EXSYST [18] is compared against GUITAR, RANDOOP, and EvoSuite [14]; Bauersfeld et al. [6] compare their ant system to random selection, but they only report on the size of the maximum call tree and not on the code coverage or faults found; GUITAR [29] does implement a random algorithm, but when compared to dynamic systems, it has been shown to perform worse [5]. Our results show that random selection is a viable option for dynamic GUI testing.

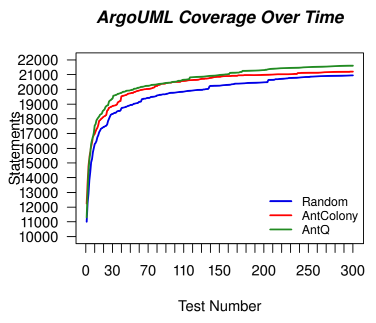

Fig. 4: ArgoUML

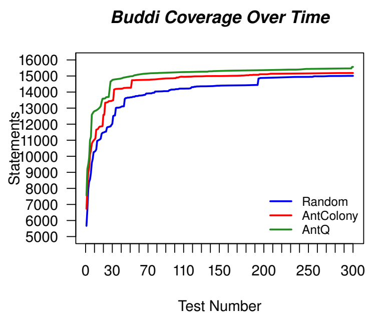

Fig. 5: Buddi

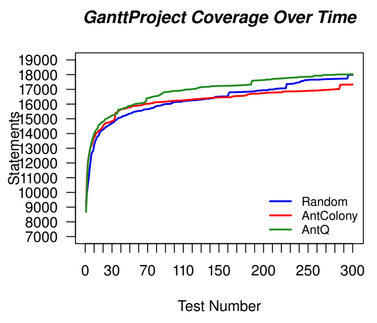

Fig. 6: Gantt Project

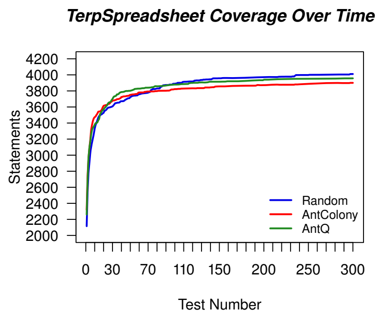

Fig. 7: TerpSpreadsheet

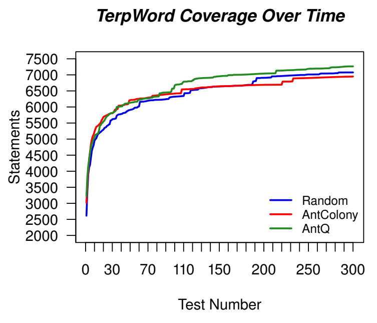

Fig. 8: TerpWord

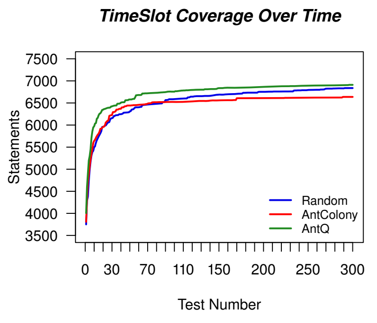

Fig. 9: TimeSlot

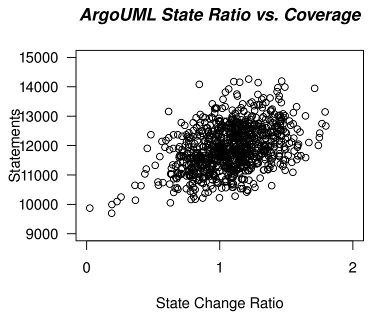

Fig. 10: ArgoUML

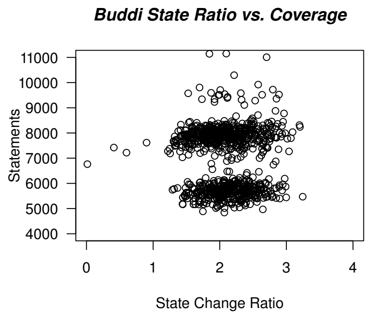

Fig. 11: Buddi

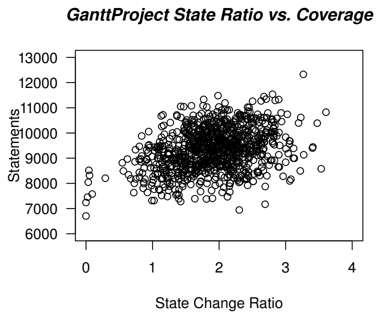

Fig. 12: Gantt Project

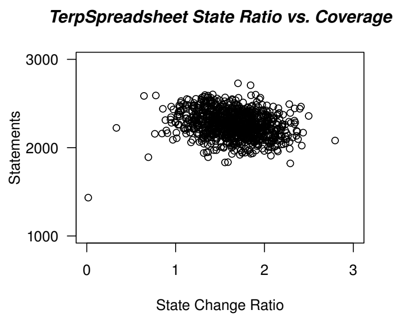

Fig. 13: TerpSpreadsheet

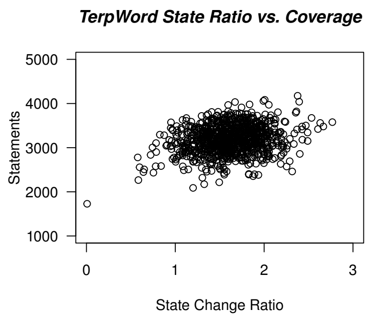

Fig. 14: TerpWord

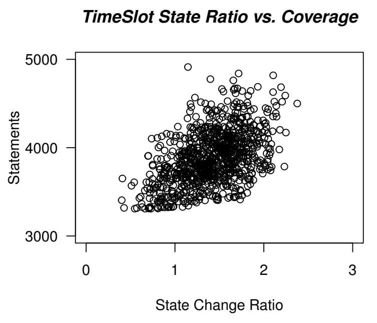

Fig. 15: TimeSlot

J. Threats to Validity

Internal validity, which refers to our ability to determine a causal relationship in our observations, in this case the algorithm used and the code coverage and exceptions found, is held by the fact that only the algorithms used could have affected the coverage metrics and exceptions. Furthermore, the applications under test have all been used in the literature and are therefore less prone to selection bias. TerpWord and TerpSpreadsheet were used in [9] [23], Gantt Project in [36] [8], ArgoUML in [29] [33], and Buddi and TimeSlotTracker were used to evaluate AutoBlackTest [22]. When testing each of the AUTs, we used the default settings provided; as well, we cleared any changes to the properties after the completion of each test case.

External validity, which refers to our ability to generalize our results, is also held. GUIs are a general concept that can be implemented in any programming language. The widgets used in most GUIs (buttons, text boxes, etc.) can be expected to work the same as the widgets provided by Java Swing. Furthermore, the concept of a GUI heirarchy that can be traveresed can also be expected to exist in other GUI implementations. There exist GUIs with a rich array of input types, such as for mobile applications, that deal with touch input. These types of inputs can usually be programatically executed and so methods such as ours should still be applicable. Therefore, the methods presented in this paper should be generalizable to other systems.

Construct validity, which refers to how we measured the results, is also upheld. The code coverage was measured using Cobertura [10], an open source application that is well maintained. We counted uncaught exceptions by implementing our own exception listener, which extends the exception listener that is a part of the JDK.

VI. Conclusion and Future Work

In this paper we introduced a new method of testing GUIs based on ant colony optimization. We created two ant systems. The first system implements the normal ant colony optimization algorithm in which ants explore a graph and pheromones are deposited after each generation. Our second algorithm implements the antq algorithm, which uses Q-Learning to update the paths as ants explore the graph. Our fitness function is based on the amount of change that occurs in the GUI as events are executed. Both ant systems use the same fitness function.

We compared our two ant systems against random selection and found that the AntQ system was able to achieve the highest levels of code coverage. When comparing the number of uncaught exceptions, we found that the three algorithms found a similar number of errors on average, and that the AntQ algorithm found the most in total. Our results merit further study into the ant system and random selection in general.

In the future we would like to incorporate the ability to evaluate correct GUI states, such as those presented in [35] as our system currently only looks at uncaught exceptions. We would also like to investigate an algorithm for finding good inputs to the GUI widgets, such as the method presented in [7], where the authors extract the widget data based on textual descriptions, either from the widget itself or surrounding widgets. This information can be used to determine what type of data to input during a test's execution, such as a digit, string, or date.

Acknowledgment

We would like to thank Leonardo Mariani and Oliviero Riganelli for their input regarding the fitness function calculation.

References

[1] SeleniumHQ: Web application testing system. http://seleniumhq.org/. Online. Accessed Feb. 2012

[2] Andrews, J.H., Briand, L.C., Labiche, Y.: Is mutation an appropriate tool for testing experiments? In: Proceedings of the 27th International Conference on Software Engineering (ICSE 2005). St. Louis, Missouri (2005). 402-411

[3] Arcuri, A., Iqbal, M.Z., Briand, L.: Formal analysis of the effectiveness and predictability of random testing. In: ACM International Conference on Software Testing and Analysis (ISSTA), pp. 219–230 (2010)

[4] Arcuri, A., Iqbal, M.Z., Briand, L.: Random testing: Theoretical results and practical implications. Software Engineering, IEEE Transactions on **38**(2), 258–277 (2012)

[5] Bae, G., Rothermel, G., Bae, D.H.: On the relative strengths of model-based and dynamic event extraction-based GUI testing techniques: An empirical study. In: Software Reliability Engineering (ISSRE), 2012 IEEE 23rd International Symposium on, pp. 181–190. IEEE (2012)

[6] Bauersfeld, S., Wappler, S., Wegener, J.: A metaheuristic approach to test sequence generation for applications with a GUI. In: Search Based Software Engineering, pp. 173–187. Springer (2011)

[7] Becce, G., Mariani, L., Riganelli, O., Santoro, M.: Extracting widget descriptions from GUIs. In: Fundamental Approaches to Software Engineering, pp. 347–361. Springer (2012)

[8] Brooks, P.A., Memon, A.M.: Introducing a test suite similarity metric for event sequence-based test cases. In: Software Maintenance, 2009. ICSM 2009. IEEE International Conference on, pp. 243–252. IEEE (2009)

[9] Bryce, R.C., Sampath, S., Memon, A.M.: Developing a single model and test prioritization strategies for event-driven software. Software Engineering, IEEE Transactions on **37**(1), 48–64 (2011)

[10] Cobertura Development Team: Cobertura web site (2010). `cobertura.sourceforge.net`

[11] Csallner, C., Tillmann, N., Smaragdakis, Y.: DySy: Dynamic symbolic execution for invariant inference. In: Proceedings of the 30th international conference on Software engineering, pp. 281–290. ACM (2008)

[12] Dorigo, M., Gambardella, L.: Ant-q: A reinforcement learning approach to the traveling salesman problem. In: Proceedings of ML-95, Twelfth Intern. Conf. on Machine Learning, pp. 252–260 (2014)

[13] Dorigo, M., Maniezzo, V., Colorni, A.: Ant system: optimization by a colony of cooperating agents. Systems, Man, and Cybernetics, Part B: Cybernetics, IEEE Transactions on **26**(1), 29–41 (1996)

[14] Fraser, G., Arcuri, A.: Evosuite: automatic test suite generation for object-oriented software. In: Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering, pp. 416–419. ACM (2011)

[15] Fraser, G., Zeller, A.: Mutation-driven generation of unit tests and oracles. In: Intl. Symp. on Software Testing and Analysis (ISSTA), pp. 147–158 (2010)

[16] Godefroid, P., Klarlund, N., Sen, K.: DART: Directed automated random testing. In: Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation (PLDI), pp. 213–223. Chicago (2005)

[17] Gross, F., Fraser, G., Zeller, A.: EXSYST: search-based GUI testing. In: Intl. Conf. on Software Eng. (ICSE), pp. 1423–1426 (2012)

[18] Gross, F., Fraser, G., Zeller, A.: Search-based system testing: high coverage, no false alarms. In: Proceedings of the 2012 International Symposium on Software Testing and Analysis, pp. 67–77. ACM (2012)

[19] Harman, M., Jones, B.: Search-based software engineering. Journal of Information and Software Technology **43**, 833–839 (2001)

[20] Huang, S., Cohen, M.B., Memon, A.M.: Repairing GUI test suites using a genetic algorithm. In: Software Testing, Verification and Validation (ICST), 2010 Third International Conference on, pp. 245–254. IEEE (2010)

[21] Jia, Y., Harman, M.: An analysis and survey of the development of mutation testing. Software Engineering, IEEE Transactions on **37**(5), 649–678 (2011)

[22] Mariani, L., Pezzè, M., Riganelli, O., Santoro, M.: AutoBlackTest: a tool for automatic black-box testing. In: Software Engineering (ICSE), 2011 33rd International Conference on, pp. 1013–1015. IEEE (2011)

[23] McMaster, S., Memon, A.M.: Call-stack coverage for GUI test suite reduction. Software Engineering, IEEE Transactions on **34**(1), 99–115 (2008)

[24] McMinn, P.: Search-based software test data generation: a survey. Software testing, Verification and reliability **14**(2), 105–156 (2004)

[25] Memon, A.M.: An event-flow model of GUI-based applications for testing. Software Testing, Verification and Reliability **17**(3), 137–157 (2007)

[26] Memon, A.M., Banerjee, I., Nagarajan, A.: GUI ripping: Reverse engineering of graphical user interfaces for testing. In: Working Conf. on Reverse Eng. (WCRE), pp. 260–269 (2003)

[27] Memon, A.M., Pollack, M.E., Soffa, M.L.: Hierarchical GUI test case generation using automated planning. Software Engineering, IEEE Transactions on **27**(2), 144–155 (2001)

[28] Myers, G.J.: The Art of Software Testing. Wiley, New York (1979)

[29] Nguyen, B., Robbins, B., Banerjee, I., Memon, A.: GUITAR: an innovative tool for automated testing of GUI-driven software. Automated Software Engineering pp. 1–41 (2013)

[30] Pacheco, C., Lahiri, S.K., Ernst, M.D., Ball, T.: Feedback-directed Random Test Generation. In: In Proceedings of the 29th International Conference on Software Engineering (ICSE 2007), pp. 75–84. Minneapolis, MN (2007)

[31] Ruiz, A., Price, Y.W.: Test-driven GUI development with testng and abbot. Software, IEEE **24**(3), 51–57 (2007)

[32] Tillmann, N., De Halleux, J.: Pex–white box test generation for. net. In: Tests and Proofs, pp. 134–153. Springer (2008)

[33] Van Rompaey, B., Du Bois, B., Demeyer, S., Rieger, M.: On the detection of test smells: A metrics-based approach for general fixture and eager test. Software Engineering, IEEE Transactions on **33**(12), 800–817 (2007)

[34] Watkins, C.J., Dayan, P.: Q-learning. Machine learning **8**(3-4), 279–292 (1992)

[35] Xie, Q., Memon, A.: Designing and comparing automated test oracles for GUI-based software applications. ACM Trans. Softw. Eng. Methodol. **16** (2007)

[36] Xie, Q., Memon, A.M.: Using a pilot study to derive a GUI model for automated testing. ACM Transactions on Software Engineering and Methodology (TOSEM) **18**(2), 7 (2008)

[37] Xie, T., Tillmann, N., de Halleux, J., Schulte, W.: Fitness-guided path exploration in dynamic symbolic execution. In: Dependable Systems & Networks, 2009. DSN'09. IEEE/IFIP International Conference on, pp. 359–368. IEEE (2009)

[38] Yuan, X., Cohen, M., Memon, A.M.: Covering array sampling of input event sequences for automated GUI testing. In: Automated Software Eng. (ASE), pp. 405–408 (2007)