ABSTRACT

GUIs are event-driven applications where the flow of the program is determined by user actions such as mouse clicks and key presses. GUI testing is a challenging task not only because of the combinatorial explosion in the number of event sequences, but also because of the difficulty to cover the large number of data values. We propose GUICat, the first cloud-based GUI testing framework that simultaneously generates both event sequences and data values. It is a white-box GUI testing tool that augments traditional sequence generation techniques with concolic execution. We also propose a cloud-based parallel algorithm for mitigating both event sequence explosion and data value explosion, by distributing the concolic execution tasks over public clouds such as Amazon EC2. We have implemented and evaluated the new tool on standard GUI testing benchmarks. Our experiments show that GUICat significantly outperforms state-of-the-art GUI testing tools such as GUITAR.

CCS Concepts

-Software and its engineering → Software verification and validation; Software testing and debugging;

Keywords

Symbolic execution, Test generation, GUI testing, Cloud

1. INTRODUCTION

Graphical User Interfaces (GUIs) provide a convenient way for the user to interact with the computer. They are event-driven applications where the flow of the program is determined by user actions such as mouse clicks and key presses. In contrast to console applications whose only point of interaction is at the beginning, GUIs have a potentially large number of interaction points, each of which may be associated with a different state. These features often make traditional software testing techniques ineffective. Specifically, GUI testing has two significant challenges. First, covering all possible event sequences of a GUI application is difficult due to the combinatorial explosion, i.e., the number of possible ways of clicking k buttons can be as large as k!. Second, GUI behaviors depend not only on the event sequence but also on the data values of widgets such as text-boxes, edit-boxes, and combo-boxes, thus leading to an extremely large input space. For example, covering all possible values of a k-character input string requires $2^k$ test cases. Although existing GUI testing tools [9–11, 13] have addressed the challenge of generating high-quality event sequences, they have not addressed the challenge of simultaneously generating high-quality data values. As such, data-dependent GUI behaviors are often inadequately tested.

We propose GUICat, a cloud-based GUI testing framework that generates both high-quality event sequences and high-quality data values, by augmenting state-of-the-art event sequence generation techniques with concolic execution. The result is a white-box GUI testing tool that uniformly explores the event flow as well as the data flow. We also propose a parallel concolic execution algorithm for mitigating the data value explosion, by distributing the computation tasks over workers on private clusters as well as public clouds such as Amazon EC2 [3]. It provides an illusion of running GUICat on a powerful super computer and thus allows it to handle significantly larger applications than previously possible.

We have implemented GUICat based on a number of open-source tools, including GUITAR [10] for generating the initial event sequences, ASM [1] for Java bytecode instrumentation, Categ [12] for concolic execution, and JaCoCo [2] for computing code coverage. Unlike prior techniques, GUICat is fully automated in modeling GUI widgets. That is, it does not require developers to manually model these widgets. This is important because manual modeling is not only labor intensive and error prone but also hard to sustain in the long run due to frequent widget updates.

We have evaluated GUICat on Amazon EC2 for a set of GUI testing benchmarks. GUICat achieves scalability through the distribution of symbolic execution tasks.

The remainder of the paper is organized as follows. We illustrate the main idea behind GUICat using motivating examples in Section 2. We present our algorithm in Section 3, which is followed by our experimental results in Section 4. We discuss the related work in Section 5. Finally, we give our conclusions in Section 6.

2. MOTIVATING EXAMPLES

Figure 1 shows a GUI example for computing ticket fare based on user inputs including Name, Age Level, Distance, and Coupons. Once the Buy button is clicked, the application computes and then displays ticket price, using a coefficient associated with the chosen age level. To allow GUICat to generate test cases, the user must provide a configuration file that specifies the name and type of the symbolic variables as shown in Figure 1 (right). Each entry (line) of the configuration file consists of the widget name, widget type, method for obtaining user input (e.g., getText), type of user input, and the default value (e.g., superman). Here, 0:1 means the default value is of enum type with two values 0 and 1.
Figure 1: A GUI example and GUICat’s configuration file.

Figure 2: Code snippet for computing the ticket price.

```java
OnClickComputePrice() {
    int coupon = 0;
    String age = (String)ageComboBox.getSelectedItem();
    String distance = distanceTextField.getText();
    int distance = Integer.parseInt(distance);
    if (d100CheckBox.isSelected())
        coupon += 100;
    else if (d200CheckBox.isSelected())
        coupon += 200;
    if (d400CheckBox.isSelected())
        coupon += 400;
    if (age.equals(Child))
        coeffienct = 1;
    else
        coeffienct = 2;
    if (distance < 60) {
        price = 500;
    } else if (distance < 80) {
        price = 11 * distance * coeffienct - coupon;
    } else if (distance < 100) {
        price = 10 * distance * coeffienct - coupon;
    } else if (distance < 120) {
        price = 9 * distance * coeffienct - coupon;
    } else {
        price = 8 * distance * coeffienct - coupon;
    }
    assert (price > 0);
    infoField.setText(price);
}
```

Figure 3: Show the architecture of GUICat. Given a GUI program P as input, GUICat first invokes GUITAR to generate event sequences. Then, it instruments the program based on each sequence and the symbolic variables specified in guicat-conf. Next, it invokes the distributed algorithm to conduct symbolic execution of the instrumented program on a cloud node. Finally, the test cases generated by all instrumented programs are collected and then used by JaCoCo to compute the coverage report.

3. ARCHITECTURE

In the distributed symbolic execution algorithm, N_0 is the load balancer and N_1, . . . , N_5 are the k workers on the cloud. N_0 distributes the set E of instrumented GUI programs, one per event sequence, over the k workers. The workers then conduct symbolic execution on their share of tasks. Initially, each worker receives roughly the same number of tasks. However, since the cost of symbolic execution varies for each event sequence, some workers may finish their symbolic execution tasks sooner than others. N_0 detects such imbalance and requests a worker with the largest workload to share its tasks with the idle worker. After all workers complete their tasks, the load balancer N_0 collects the test cases.

Algorithm 1 shows the symbolic execution procedure for each individual worker. Initially, the set E of tasks and test sequences end in e7 because otherwise no action can be taken at the end of the user interaction. Consider (e_2, e_7) as an example. It means the user specifies a distance (e_2) before clicking the BuyButton (e_7). Although logically meaningless, this particular event sequence is feasible.
Algorithm 1 Symbolic Execution on a Worker.

1: \( E = T = \emptyset \)
2: \( \textbf{while true do} \)
3: \( \text{if receiving task set } E' \text{ then} \)
4: \( \quad E = E \cup E' \)
5: \( \quad \text{send}(N_0, |E|); \)
6: \( \text{else if requesting tasks on behalf of } N_0 \text{ then} \)
7: \( \quad \text{send}(N_0, |E|/2); \)
8: \( \quad \text{send}(N_0, |E|); \)
9: \( \text{else if collecting test cases then} \)
10: \( \quad \text{send}(N_0, T_0); \)
11: \( \quad \text{terminate}; \)
12: \( \textbf{end if} \)
13: \( \textbf{for all } e \in E \textbf{ do} \)
14: \( \quad P = \text{instrument}(P, e, c/fy) \)
15: \( \quad T_e = \text{Catg'.execute}(P); \)
16: \( \quad \text{send}(N_0, |E|); \)
17: \( \quad T = T \cup T_e; \)
18: \( \textbf{end for} \)
19: \( \textbf{end while} \)

cases \( T \) are both empty. Then, the worker keeps checking messages from \( N_0 \) and conducting local symbolic execution. If it receives a set \( E' \) of tasks from \( N_0 \) or another worker, the new tasks are added to the local set. Since the number of tasks is changed, it updates \( N_0 \) with its current number of tasks. This also occurs at Lines 8 and 16. With such updates, \( N_0 \) knows the number of tasks to be processed at the workers. If the worker receives a message from \( N_0 \) that requests more tasks on behalf of another worker \( N_0 \), it sends half of its tasks to \( N_0 \) (Line 7). A signal of termination is received if \( N_0 \) asks for its test cases, and in such case, the current worker sends the locally generated test cases and then terminates.

Symbolic execution is conducted for each individual event sequence \( e \in E \). Essentially, it allows us to execute the event-driven application as if it is a sequential Java program. We leverage the Catg concolic execution tool, which maintains two execution stacks: one for concrete execution and the other for symbolic execution. When Catg executes an unknown method, for example, \( \text{Integer.parseInt()} \), the symbolic execution stack would not be updated, we have modified Catg to handle unknown methods.

The distributed algorithm in GUICat has been implemented on Amazon EC2 as a Cloud service. We divide EC2 instances into the load balancer and the workers. The load balancer is a multi-processor EC2 instance that generates event sequences, distributes tasks to the workers, and collects the test cases generated by the workers. Each worker is a single-processor EC2 instance that symbolically executes an event sequence to generate test cases. GUICat allows the user to customize the Cloud service, such as the number of workers and their computation capabilities, based on customer requirements such as whether a budget-first testing is preferred over a speed-first testing, or whether branch coverage is preferred over instruction coverage.

To allow easy customization, we implement GUICat by following the star topology shown in Figure 4, where the load balancer generates and distributes event sequences to the workers, and the workers conduct concolic execution with respect to the event sequence in isolation, before sending test cases back to the load balancer. Figure 5 illustrates how event sequences are distributed to the workers and how test cases are collected from the workers. For now, distributed file system libraries are used to implement the transfer of event sequences and test cases between the load balancer and workers. The main advantage of this architecture is efficiency since there is no communication between the workers.

4. EXPERIMENTAL EVALUATION

We have implemented GUICat using a number of open-source tools, including GUITAR for generating the initial event sequences, ASM for instrumenting the Java bytecode, Catg for implementing the distributed parallel concolic execution, and JaCoCo for computing the code coverage report.

We have evaluated GUICat on several GUI testing benchmarks. In all experiments, we have used the Amazon EC2 cloud computing infrastructure, where the load balancer is deployed as a multi-processor EC2 instance and each worker is deployed as a single-processor EC2 instance.

In the remainder of this section, we report the results of two case studies: a ticket seller and a workout generator. In each case study, our experiment consists of the following steps. First, we create a configuration file for the application under test. Then, we generate the event sequences using GUITAR [10]. Next, we distribute the event sequences from the load balancer to workers on Amazon EC2. The initial distribution is static and divides the tasks evenly to the EC2 instances. After receiving the event sequences, each worker conducts concolic execution using Catg; as a result, test cases...
are generated for these event sequences. When all workers finished, the load balancer collects their test cases and then uses JaCoCo to compute the coverage report.

### 4.1 The Ticket Seller

Figure 6 shows the user interface of a more sophisticated ticket seller than the one shown in Figure 1. It allows the user to provide passenger information such as the Name, ID, start distance (From), end distance (To), Age Level, Class Level, and the Coupon. When the user clicks the Buy Ticket button, the application stores the passenger information to an object named TicketModel, checks for consistency using the method checkModel(), and computes the price using the method computePrice().

There are five different types of GUI widgets in Figure 6: four of JTextField type, one of JComboBox type, one of JRadioButton type, one of JCheckBox type, and two of JButton type.

The first two JtextField widgets collect the values of Name and ID by invoking getText(), the combination of which may lead to buggy behaviors. We mark both fields as symbolic. That is, when loading the related Java class, we use the bytecode rewriting tool ASM to instrument the program on the fly, to replace invocations of getText() with invocations of sGetText(), a method that we develop to return a symbolic value. The symbolic values for Name and ID are used during the subsequent symbolic execution step.

The next two JtextField widgets collect the values of From and To by first invoking JtextField.getText() and then invoking Integer.parseInt() to cast the strings into integers. Since the symbolic execution engine Catg does not support such casting, we have modified Catg to convert these strings into integers before using them in the subsequent logic.

As for the selectable widgets JComboBox, and JCheckBox, we enumerate all possible values of the enum type. We choose enumeration over symbolic execution for these selectable widgets due to efficiency and ease of implementation. First, selectable widgets do not have many different values. Second, the existing symbolic execution engine often has trouble handling them. For example, JComboBox has two methods getSelectedindex() and getSelecteditem() that return values of a customized Object type, which cannot be easily cast to strings or integers inside Catg.

To summarize, we used GUITAR to generate the initial event sequences together with the initial parameters/states. Then, we add the enumerated values for selectable widgets, before conducting symbolic execution to generate the values for widgets of other types. If the initial event sequence is too short to contain all widgets of JTextField type needed, we remove the stateless JTextField event and then append more stateful JTextField event to the beginning of the event sequence, thus increasing the length of the sequence.

### 4.2 The Workout Generator

Figure 9 shows the user interface of the workout generator, which is taken from Barad [6]. It generates a workout plan based on the user input, including the Gender, Metabolism, Experience, Age, Height, and Weight. The computation starts when the user clicks the Generate button. Depending on the

<table>
<thead>
<tr>
<th>Tool</th>
<th>Button</th>
<th>Branches</th>
<th>Coverage</th>
<th>Instructions</th>
<th>Coverage</th>
</tr>
</thead>
<tbody>
<tr>
<td>GUITAR</td>
<td>checkModel</td>
<td>6</td>
<td>33.3%</td>
<td>56</td>
<td>53.3%</td>
</tr>
<tr>
<td>GUITAR</td>
<td>computePrice</td>
<td>34</td>
<td>23.5%</td>
<td>172</td>
<td>30.8%</td>
</tr>
<tr>
<td>GUITAR</td>
<td>checkModel</td>
<td>6</td>
<td>100%</td>
<td>56</td>
<td>100%</td>
</tr>
<tr>
<td>GUITAR</td>
<td>computePrice</td>
<td>34</td>
<td>100%</td>
<td>172</td>
<td>100%</td>
</tr>
</tbody>
</table>

We analyzed the ticket seller with several configurations. Table 1 shows the results of generating the length-2 test cases. Columns 1-2 show the tool name and the name of the button clicked. Columns 3-4 show the total number of branches and the percentage of branches covered. Columns 5-6 show the total number of instructions and the percentage of instructions covered. The results show that GUITAR can achieve full branch and instruction coverage even with length-2 test cases, whereas GUITAR can only achieve 33.3% branch coverage and 53.5% instruction coverage for checkModel, 23.5% branch coverage and 30.8% instruction coverage for computePrice.

Table 2 compares GUITAR and GUITAR in terms of the number of paths covered. Column 1 shows the tool name. Column 2 shows the length of the test sequences. Column 3 shows the total number of test cases generated by GUITAR. Column 4 shows the number of additional test cases generated by GUITAR after enumerating the values of selectable widgets. Column 5 shows the number of test cases generated by GUITAR after concolic execution. Column 6 shows the path coverage.

To accurately compute the number of paths covered, we manually added code into the program. Specifically, we used a vector named path, where each element was mapped to an if-statement. For example, the age combo-box corresponds to an if-statement where we set path[0]=0 in the then branch and path[0]=1 in the else branch. Each time the program terminates, we will obtain a unique vector path that acts as the path identifier. These vectors are stored and then used to compute the path coverage after GUITAR terminates.

Our result shows that, overall, GUITAR achieved significantly higher path coverage than GUITAR. For length-2 test sequences, in particular, GUITAR had 96/4=24 times higher path coverage, whereas for length-3 test sequences, GUITAR had 190/7=27 times higher path coverage.

GUITAR also successfully detected two bugs in ticket seller. One bug is a NullPointerException caused by the race condition between clicking of the save button and clicking of the buy button as shown in Figures 7 and 8. The other bug is the failure of an assertion due to the computed price being less than zero.
user information, the computation goes through different execution paths that use different cardio coefficients.

The reason is that branch/instruction coverage. The reason is that branch/instruction coverage. The reason is that branch/instruction coverage. The reason is that branch/instruction coverage. The reason is that branch/instruction coverage. The reason is that branch/instruction coverage. The reason is that branch/instruction coverage. The reason is that branch/instruction coverage.

The text box contains the buggy code snippet in ticket seller.

The text box contains the buggy code snippet in ticket seller.

The table shows the results of applying GUICat to workout generator (node=1). The table compares GUICat and GUITAR in terms of the number of paths covered. Column 1 shows the tool name. Column 2 shows the length of the test sequences used in the experiments. Column 3 shows the number of test cases generated by GUICat. Column 4 shows the number of additional test cases generated by GUICat after enumerating the values of selectable widgets. Column 5 shows the number of test cases generated by GUICat after concolic execution. Column 6 shows the number of paths covered. Our result demonstrates that, overall, GUICat can achieve significantly higher path coverage than GUITAR. For length-2 test sequences, in particular, GUICat reached 24/1=24 times higher path coverage, whereas for length-3 test sequences, GUICat reached 56/2=28 times higher path coverage.

We also observed that GUICat generated many more test cases than the paths covered. For example, with the length set to 3, GUICat generated 909 test cases to cover 56 unique paths, which means some of these test cases have led to the same paths. If we could, for example, identify and eliminate these redundant test cases, the performance of GUICat would be further improved. However, we leave the pruning of these redundant test cases for future work.

4.3 Effect of Cloud Computing

Figure 10 shows the effectiveness of distributing the testing of the ticket seller (a) and the workout generator (b) over Amazon EC2. The x-axis denote the number of workers, ranging from 1 to 16, and the y-axis denote the time usage in second. The solid, dashed and dotted lines represent the bounded length of 2, 3 and 4, respectively. Due to the inherent parallelism in symbolic execution of different event sequences, the speedup is almost linear.

5. RELATED WORK

GUITAR [10] is the first framework capable of performing the whole process of test generation, execution, and result assessment for GUIs. Since its first publication there have been multiple improvements (e.g. [14]). This framework generates tests as event sequences up to a given bound. For emulating user input a specification based approach is adopted, i.e., using values from a prefiled database. Since GUITAR does not provide a mechanism for reasoning about input values for data widgets, GUICat offers complementary and more comprehensive testing.
The work closest to ours is Barad [6] that also exploits symbolic execution to compute input values for data widgets. It manually creates symbolic mirror of Java GUI library, so its released source code contains a large symbolic Java GUI library. Manual modeling is error prone and hard to sustain. In fact, we downloaded the tool but failed to make it work on our benchmarks. We employ a different test generation algorithm and symbolic analysis method for obtaining inputs. Another line of work is to apply model checking techniques. For example, jfp-awt [8] is an extension of the Java PathFinder for GUI applications.

Performance enhancement of GUI testing has traditionally focused on minimizing event sequences [9, 13]. Barad generates test cases as chains of event listener method invocations and maps these chains to event sequences that force the execution of these invocations. Such approach prunes the event input space because it does not need to consider events where there are no event listeners. More recent work starts to apply event dependency analysis [4], program slicing [5] or partial order reduction [7] to improve the performance. Our performance improvement is obtained by exploiting massive hardware resource available on cloud. Therefore our approach is orthogonal to the existing algorithmic approaches.

6. CONCLUSION

We have present GUICat, the first cloud-based GUI testing tool for simultaneously generating high-quality event sequences as well as high-quality data values. Internally, GUICat leverages GUITAR to generate the initial set of event sequences, and then uses a combination of value enumeration and symbolic execution to generate data values of the widgets. GUICat also leverages the cloud computation infrastructure to speed up the test generation, by distributing independent concolic execution tasks to EC2 nodes. We have implemented GUICat and evaluated it on a set of GUI testing benchmarks. Our experiments show that GUICat can significantly outperform GUITAR on standard GUI testing benchmarks in terms of both branch coverage and instruction coverage.

7. ACKNOWLEDGMENTS

This work was primarily supported by NSF under grants DGE-1522883 and CCF-1500365. Partial support was provided by CCF-1149454, CCF-1405697, and CCF-1500024.

References