Systematic Reduction of GUI Test Sequences

Lin Cheng, Zijiang Yang
Western Michigan University
Kalamazoo, MI, USA

Chao Wang
University of Southern California
Los Angeles, CA, USA

***Abstract*—Graphic user interface (GUI) is an integral part of many software applications. However, GUI testing remains a challenging task. The main problem is to generate a set of high-quality test cases, i.e., sequences of user events to cover the often large input space. Since manually crafting event sequences is labor-intensive and automated testing tools often have poor performance, we propose a new GUI testing framework to efficiently generate progressively longer event sequences while avoiding redundant sequences. Our technique for identifying the redundancy among these sequences relies on statically checking a set of simple and syntactic-level conditions, whose reduction power matches and sometimes exceeds that of classic techniques based on partial order reduction. We have evaluated our method on 17 Java Swing applications. Our experimental results show the new technique, while being sound and systematic, can achieve more than 10X reduction in the number of test sequences compared to the state-of-the-art GUI testing tools.**

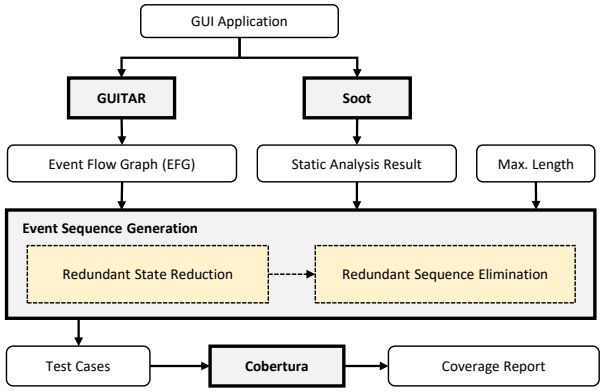


Fig. 1. Systematic generation and reduction of GUI tests.

I. Introduction

Graphic user interface (GUI) is an integral part of many software applications that monitor user actions such as keyboard and mouse events and respond by invoking listener functions. To test a GUI application, one must create tests to cover its input space, where each test is a finite sequence of events. Due to combinatorial blowup, the number of sequences can be astronomically large, e.g., up to 10^{10} for all length-10 sequences of 10 events, if these events are enabled all the time. Thus, the main problem is to generate a small subset of these event sequences while achieving the same testing effect as the complete set. Since manually crafting these sequences is labor-intensive, techniques have been developed to generate them automatically [57], [56], [40], [2], [3], [54]. Unfortunately, these existing techniques are neither *systematic* nor *efficient*, i.e., they often miss important event sequences and produce many redundant sequences.

To avoid these problems, we propose a new test generation tool to construct progressively longer event sequences. Our tool has the advantage that, during the sequence generation process, it eliminates an event sequence only if the sequence is guaranteed to be redundant, i.e., subsumed by some other sequences. This is accomplished by a new type of reduction technique that differs from classic partial order reduction (POR) methods [49], [41], [19], [18], [15]. Our tool is also efficient in that it relies on a set of easily-checkable conditions to identify redundant sequences. These conditions are expressed in terms of the sequence of events as opposed to the concrete program states. Thus, they can be checked by a purely static analysis of the event flow of the GUI application, without executing the actual application.

Fig. 1 shows the overall flow of our method. The input consists of Java byte-code of the GUI application and a bound on the sequence length. The output is a set of event sequences. Internally, our method goes through several steps. First, it leverages GUITAR [21] to reverse-engineer an event flow graph (EFG) of the application. The EFG shows the set of events enabled at any step of the execution, as well as the enabled events afterward. Then, our method leverages Soot [48] to perform static analysis of the Java byte-code to compute dependencies over the events. Next, it invokes our core algorithm, which takes the EFG, the dependencies and a bound as input and constructs the test sequences. Finally, the sequences are executed on the actual GUI application using Cobertura [12], which measures the coverage.

Compared to state-of-the-art GUI testing tools [3], [2], [40], our method has two advantages. First, it is systematic, meaning that useful test sequences are not excluded in any *ad hoc* fashion: within the maximum sequence length, our method eliminates a sequence only if it is provably redundant; when in doubt, it retains the sequence. Second, our method is efficient in that it generates significantly fewer test sequences than prior techniques. For instance, when applied to the example in Fig. 2, prior techniques based on partial order reduction can only remove 11 redundant sequences, whereas our method removes 34 redundant sequences. Although Arlt et al. [3] proposed a reduction technique that goes beyond POR, their tool still generates significantly more sequences for the example in Fig. 2: when the maximum length is set to 5, 7 and 9, it generates 33, 129 and 513 sequences, respectively, whereas our method generates only 6 sequences.

We have evaluated our method on 17 Java Swing applications consisting of 105,937 lines of code in total. The experimental results show our new reduction technique is more effective: it outperforms partial order reduction consistently

978-1-5386-2684-9/17 © 2017 IEEE

ASE 2017, Urbana-Champaign, IL, USA
Technical Research

```
1   class ModifyImageWindow extends JFrame {
2     boolean convert = false;
3     int angle = 0;
4     void onCheckBox() {
5       int cbValue = checkBox.getValue();
6       convert = (1 == cbValue) ? true : false;
7     }
8     void onSlider() {
9       int sliderValue = slider.getValue();
10      angle = sliderValue;
11      print(convert, angle);
12    }
13    void onSave() {
14      int anValue = angle;
15      if (anValue > 0 )
16         UserSettings.RotationAngle = anValue;
17      else
18         assert(0); //BUG#1: Crash if reached
19    }
20    void onOK() {
21      if ( convert ) {
22        image.convertToGrayscale();
23        image = null;
24      }
25      if ( angle > 0 )
26        image.rotate(angle); //BUG#2: if image==null
27      else
28        image.draw(); //BUG#3: if image==null
29    }
30  }
```

Fig. 2. The class ModifyImageWindow defines event handlers onCheckBox, onSlider, onSave, and onOK.

and significantly. We also experimentally compared with state-of-the-art GUI testing tools including Gazoo [3], [2] and GUITAR [40]. Overall, our tool achieves more than 10X reduction in the number of test sequences and significantly reduces the corresponding test execution time.

To summarize, this paper makes the following contributions:

- We propose an automated GUI testing framework for generating event sequences efficiently.
- We develop a reduction technique to more effectively eliminate redundant sequences than prior techniques.
- We use realistic applications to demonstrate the advantages of our method over state-of-the-art testing tools.

In the remaining sections, we use motivating examples to illustrate the main ideas behind our method before formally presenting the algorithm and our experimental results.

II. Motivation

Consider the Java code in Fig. 2, which controls a window that allows the user to modify an image by clicking the check box, choosing an angle from the slider control, and clicking the `OK` button. Clicking the `OK` button closes the window and thus disables all event handlers. Optionally, the user may click the `Save` button to store the angle. Fig. 3 shows the event flow graph (EFG), where nodes are events and edges indicate the set of events enabled in each step. All four events are enabled initially. However, since clicking `OK` closes the window, the node labeled `OK` has no outgoing edges.

To test all possible behaviors, we must visit all reachable states and, from each state, invoke all enabled events at least once. Naively, this can be accomplished by enumerating all event sequences in the EFG up to a predefined length.

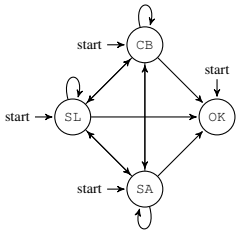


Fig. 3. Event flow graph, where `CB` , `SL` , `SA` , and `OK` denote `onCheckBox`, `onSlider`, `onSave`, and `onOk`, respectively.

A. Naive Solution

For the example in Fig. 2, all possible states will be reached after invoking at most two events and then from each state, invoking all enabled events will cover the $(state \times event)$ combinations. When the maximum sequence length is set to 3, the number of event sequences will be $(3\times3\times4+3+1=40)$ as shown in Fig. 4. The number is less than $(4\times4\times4=64)$ because clicking the `OK` button ends the execution.

However, some of these sequences are redundant. For example, $\{$`SA`, `SA`, `OK`$\}$ covers the same behavior as $\{$`SA`, `OK`$\}$: they visit the same states and, from these states, they execute the same events. This is because executing `SA` does not change the program state. Here, states are value combinations of the variable `convert` and the predicate `(angle>0)`. Although `angle` is an integer variable, the only thing matters in this application is whether `(angle>0)`. Thus, there are four distinct states: 00, 01, 10, and 11, where 00 is the initial state.

Fig. 5 shows the state transition graph (STG) where nodes are states and edges are events executed at the source states. Specifically, from the state 00, if we execute `SL`, the program goes to the state 01. From the state 01, if we execute `CB`, the program goes to the state 11. From the state 11, if we execute `CB` again, the program goes back to the state 01, because clicking `CB` twice, or any even number of times, reset the status of the check box. Finally, at any of these four states, if we click `OK`, the execution ends – in this sense, `OK` can only appear at the end of an event sequence.

B. Our New Method

As we have mentioned, ideally, we would like to execute each enabled event (`CB`, `SL`, `SA`, or `OK`) at every reachable state (00, 01, 10, and 11). Surprisingly, to achieve this goal, only a small subset of event sequences in the search tree of Fig. 4 need to be explored, as shown by the reduced tree in Fig. 6. The yellow states are irredundant states, solid blue lines are the irredundant sequences, while blue states represent the backtracking points because they match some previously explored states and thus do not need to be explored again.

Initially, the program is at state 00. State 01 can be reached via the sequence $\{$`SL`$\}$, 10 via $\{$`CB`$\}$, and 11 via $\{$`CB`, `SL`$\}$. Thus, we have brought the application to all four states. Next, we execute each enabled event at every reachable state. The number of sequences is not 16, but 13, because some of the shorter sequences are subsumed by longer ones. Specifically, there is no need to execute $\{$`CB`$\}$, $\{$`SL`$\}$, or $\{$`CB`, `SL`$\}$ from

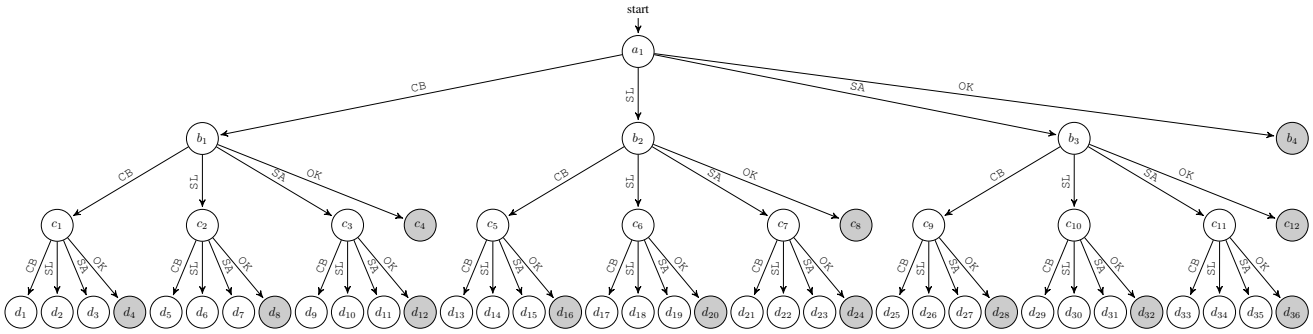


Fig. 4. The complete tree of 40 event sequences of length ≤ 3 for the GUI application in Fig. 2 and Fig. 3.

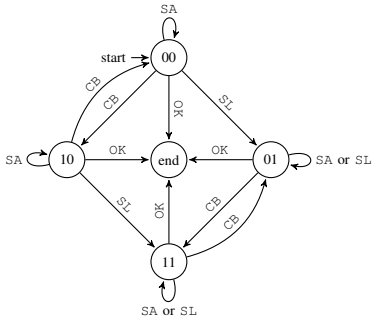


Fig. 5. State transition graph, where each state is a valuation of variable `convert` and predicate `(angle>0)`.

the initial state 00, because they are already part of the longer sequences $\{\texttt{CB}, \texttt{SL}, \texttt{OK}\}$ and $\{\texttt{SL}, \texttt{OK}\}$.

However, there are redundancies even among these 13 event sequences. For example, since `CB` does not read from any of the two variables (`convert` and `angle`), executing `CB` from any of the four states would result in the same code coverage for the listener function of `CB`. Since `CB` is enabled at state 00, there is no need to test `CB` at another state.

Similarly, since `SA` reads only from `angle`, executing `SA` from 00 and 10 (or 01 and 11) would result in the same coverage for the listener function of `SA` . Although `SL` reads from both variables, it overwrites `angle` before reading it, and thus depends only on the value of `convert`. Due to this reason, executing `SL` from 00 and 01 (or 10 and 11) would result in the same coverage for the listener function of `SL` . Finally, since `CB` depends on both variables, it has to be executed from all four reachable states.

Using this new notion of reduction, we can generate the following 6 event sequences while maintaining the same test coverage as the complete set of 40 sequences.

- $t_1 = \{\texttt{CB}, \texttt{SL}, \texttt{OK}\}$
- $t_2 = \{\texttt{CB}, \texttt{OK}\}$
- $t_3 = \{\texttt{SL}, \texttt{SA}\}$
- $t_4 = \{\texttt{SL}, \texttt{OK}\}$
- $t_5 = \{\texttt{SA}\}$
- $t_6 = \{\texttt{OK}\}$

C. Comparison to Existing Techniques

Our reduction differs from techniques based on partial order reduction (POR), which is a widely used idea for state-space reduction, e.g., in model checking [49], [41], [19], [51], [30] and concurrency testing [15], [50], [31], [16], [32], [59]. The idea of POR has also been used to reduce the cost of testing event-driven programs [33], [46], [34]. That is, when two sequences of events are equivalent permutations of each other, only one of them will be tested. However, since POR relies solely on the theory of equivalent traces [18], it can only identify redundancy in event sequences of the same length.

In contrast, our reduction goes beyond equivalent permutations; it also can identify redundancy in sequences of different lengths, e.g., as shown by $\{\texttt{SA}, \texttt{SA}, \texttt{OK}\}$ and $\{\texttt{SA}, \texttt{OK}\}$, which are not permutations of each other. Indeed, applying POR to the example in Fig. 2 would produce 29 sequences, significantly more than the 6 sequences produced by our method.

Compared to state-of-the-art GUI testing tools such as GUITAR [40] and Gazoo [3], our method also has two advantages: it does not skip useful test sequences and it often leads to fewer test sequences. Both GUITAR and Gazoo skip test sequences in an *ad hoc* manner to reduce their computational overhead, which means they often miss important corner cases. For instance, below are the seven sequences (t'_1 to t'_7) generated by Gazoo for our running example:

- $t'_1 = \{\texttt{CB}, \texttt{SL}, \texttt{OK}\}$ – same as our t_1
- $t'_2 = \{\texttt{CB}, \texttt{CB}, \texttt{OK}\}$ – equivalent to prefix t_1 : $\{\texttt{CB}, \ldots\}$ and t_6: $\{\texttt{OK}\}$
- $t'_3 = \{\texttt{SL}, \texttt{SL}, \texttt{OK}\}$ – equivalent to our t_4: $\{\texttt{SL}, \texttt{OK}\}$
- $t'_4 = \{\texttt{CB}, \texttt{CB}, \texttt{SL}\}$ – equivalent to t_3: $\{\texttt{SL}, \ldots\}$
- $t'_5 = \{\texttt{CB}, \texttt{SL}, \texttt{SL}\}$ – equivalent to t_1: $\{\texttt{CB}, \texttt{SL}, \ldots\}$
- $t'_6 = \{\texttt{SL}, \texttt{SL}, \texttt{SL}\}$ – equivalent to t_3: $\{\texttt{SL}, \ldots\}$
- $t'_7 = \{\texttt{SL}, \texttt{SL}, \texttt{SA}\}$ – equivalent to our t_3: $\{\texttt{SL}, \texttt{SA}\}$

All sequences generated by Gazoo are subsumed by our sequences. Some are clearly redundant: both t'_6 and t'_7 are subsumed by our t_3, and both t'_1 and t'_5 are subsumed by our t_1. In addition, our sequences are not only fewer (6 versus 7) but also shorter, which may translate to faster test execution.

Second, the sequences generated by Gazoo does not cover all behaviors. In particular, they missed $t_2 = \{\texttt{CB}, \texttt{OK}\}$ and $t_5 = \{\texttt{SA}\}$, both of which are useful test cases. For example, Bug#1 in Fig. 2 (Line 18) can be reached by $\{\texttt{SA}\}$ and Bug#3 (Line 28) can be reached by $\{\texttt{CB}, \texttt{OK}\}$. Since Gazoo failed to generate these two sequences, it missed these bugs.

A more severe problem of Gazoo and other existing techniques is that as the length increases, the number of sequences grows exponentially. For our running example, when the maximum sequence length is set to 5, 7, 9, ..., the number of sequences generated by Gazoo would be 33, 129, 513, ..., respectively, as shown in Fig. 7, whereas the number of sequences generated by our new method would remain 6. The

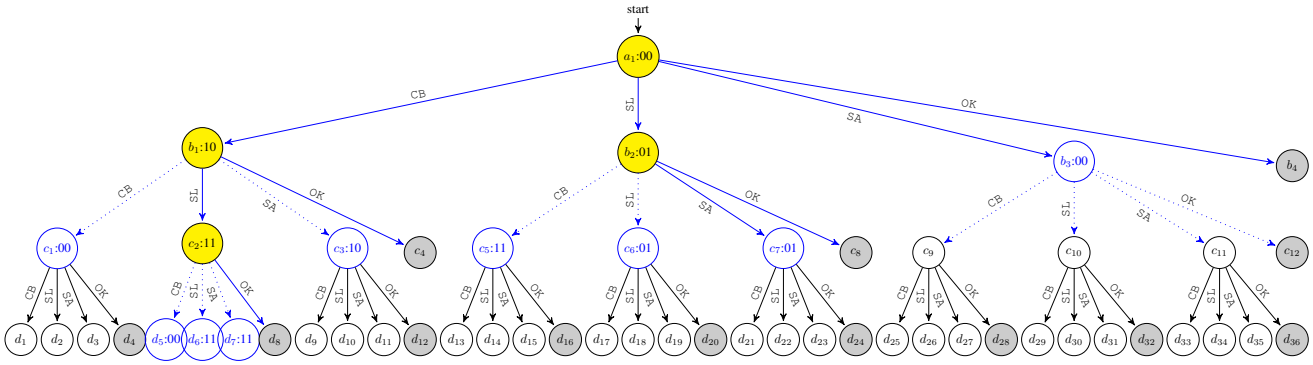


Fig. 6. The reduced tree of event sequences of length ≤ 3: From 40 sequences to 13 sequences and then to 6 sequences.

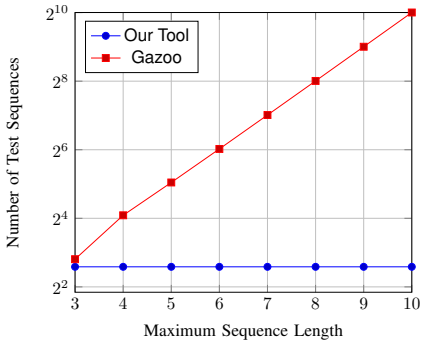


Fig. 7. The number of sequences generated by Gazoo and our tool as the maximum sequence length increases.

reason is because our six sequences already cover all possible behaviors of this GUI application as shown in Fig. 6.

D. Stateless Implementation

At this moment, one may have the impression that our new method relies on recording the states of the GUI application at run time, which is often how classic POR techniques are implemented in model checkers. However, this is not the case. Our main contribution is using *stateless* techniques to identify the provably redundant sequences without executing the GUI application. This is important because recording concrete states may be prohibitively expensive in practice.

Thus, we assume the state transition graph (STG) shown in Fig. 5 is not available. Instead, we rely on checking a set of sufficient conditions under which two sequences are guaranteed to result in the same state. Furthermore, we make sure that these conditions can be checked statically by inspecting only the event sequences and event listener functions.

For example, since `SA` does not write to any variable, executing it does not change the state. Therefore, if an event leading to the state s is `SA` (or any other event that does not modify state variables) we know s is an already-explored state. Consequently, we can skip all event sequences starting from s. Similarly, executing $\{\texttt{SL}, \texttt{SL}\}$ would lead to the same state as executing $\{\texttt{SL}\}$, assuming that executing `SL` always returns the same set of non-0 values for `angle`.

In both cases, we need not access the values of program variables. Instead, we check, for each event, what are the variables it reads from or writes to, and for any two events, if they are causally dependent (details in Section V).

These statically-checkable conditions are sufficient in that, if they hold, the corresponding sequences are guaranteed to be redundant. In this sense, our reduction never removes useful sequences. However, these conditions are not necessary: we do not attempt to capture all redundant sequences because the overhead would be prohibitively high. Inherently, this is a trade-off between the pruning capability and the computational overhead. Thus, it is important to assess *how well* our method perform in practice, e.g., does it reach or come close to the ideal reduction? In our experimental evaluation (Section VI), we will demonstrate that our carefully designed sufficient conditions work well in practice.

III. Preliminaries

A. GUI Application

A GUI application consists of (1) a set $\mathcal{C}$ of *containers*, such as windows, panels and tabs, (2) a set $\mathcal{W}$ of *widgets*, such as labels, buttons, and check boxes, and (3) a set $\mathcal{L}$ of *listener functions* associated with the widgets. Widgets are grouped by containers that host them, and are elements of interaction allowing the user to interact with functional parts of the software code. GUI libraries such as Java AWT/Swing contain a large collection of widgets and the default program logic for manipulating them.

A listener $l \in \mathcal{L}$ is a function that may be invoked to respond to a user event. Some listeners are *built-in* listeners provided by GUI libraries while others are custom-made: they are written by the application developers. For ease of presentation, we consider one listener per widget in the remainder of this paper. Thus, triggering an *event* means executing the listener associated with the *widget*.

B. Event Flow Graph

The mapping between widgets and listeners of a given application may be reverse engineered using tools such as GUITAR [21], which leverages dynamic execution and the Java *accessibility* feature to traverse an object and its children and execute their listeners. The widget-to-listener mapping obtained in this way is represented by an event flow graph

(EFG), which shows the set of events enabled at every step of the execution. The EFG is one of the inputs to test sequence generation tools including ours.

Formally, an event flow graph is a directed graph $G_{EFG} = (E, T)$, where E is a set of events and T is a set of transitions between these events. Let $E_0 \subseteq E$ be the set of events enabled at the beginning of the GUI execution. From these initial events, each subsequent transition $(ev_i, ev_j) \in T$, where $ev_i, ev_j \in E$, represents the fact that executing ev_i allows ev_j to be executed in the next step.

Fig. 3 shows an example EFG consisting of four events, which corresponds to the listener functions in Fig. 2. All four events are enabled initially, and after executing any of the first three events, all four events remain enabled. However, `OK` is different in that executing this event would end the execution: in the EFG, `OK` does not have outgoing edges.

C. Dependency Relation

Since event listener functions may read from or write to shared variables, they may impose dependency over events. In partial order reduction [18], two events are considered *conflict-dependent* (or simply *dependent*) if they access the same variable and at least one of the accesses is a write. In Fig. 2, for example, `OK` depends on `CB` and `SL` because it reads from `convert` and `angle` written by the other two events. Based on this notion of dependency, two event sequences are equivalent if they can be transformed to each other by repeatedly swapping the *adjacent* and *independent* events.

Although this dependency relation has been widely used in model checking and concurrency testing, it is often not accurate enough for dealing with events in GUI applications. For example, in Fig. 2, one `SL` event and another `SL` event have overlapping read-write and write-write sets – since they both read from and write to `angle`. However, the listener function of `SL` always overwrites the value of `angle` before reading from it, which means the behavior of the second `SL` event's listener function does not depend on the value of `angle` written by the first `SL` event. In this sense, we say these two events are *not causally* dependent.

Causal Dependency. We rely on the refined notion of dependency, namely *causal dependency*. Here, two events $ev_1, ev_2 \in E$ are causally dependent, denoted $(ev_1, ev_2) \in R_{CD}$, where R_{CD} is the dependency relation, if the execution of any one of them may affect the subsequent execution of the other. When two events are not causally dependent, we say they are *causally independent*. Causal-dependency is more accurate than conflict-dependency in that it reflects the actual impact of one event over another. In Section V, we shall explain how a simple static analysis of the event listener functions can help determine whether two events are causally dependent.

IV. Systematic Test Generation

We first present the baseline algorithm for generating test sequences (with no reduction) and then discuss how to integrate POR-based reduction into the algorithm.

A. The Baseline Algorithm

Given the EFG $G = (E, T)$ and the causal dependency relation R_{CD}, Algorithm 1 (excluding Lines 9 and 12) generates all possible event sequences up to a predetermined length. Following the notation established in stateless model checking [18], we use a stack named S to store the sequence of (abstract) states. S contains the initial state s_0 at the beginning. For each state $s \in S$, we use $s.enabled$ to denote the set of events enabled at s, use $s.selEV$ to denote the event chosen to execute at s, and use $s.done$ to denote the set of all previously chosen events at s.

The procedure EXPLORE first checks if the execution has ended, i.e., if $S.size > MAXLENGTH$ or $s.enabled = \emptyset$. If either condition is met, the while-loop would be skipped. Then, OUTPUTSEQUENCE(S) is invoked to print the event sequence stored in S, provided that S holds a complete execution (indicated by $s.selEV = NULL$) as opposed to the prefix of a longer execution. Otherwise, it enters the while-loop to execute a previously unexplored $event$, set $s.selEV = event$, and invoke EXPLORE recursively. After all events in $s.enabled$ are explored, it exits the while-loop. At this moment, $s.selEV$ will not be NULL, meaning S holds the prefix of a longer sequence (that has been printed).

Consider the running example in Fig. 2. Applying Algorithm 1 with MAXLENGTH=3 would explore the complete tree of 40 sequences as shown in Fig. 4. Clearly, some of these sequences are redundant and thus should be removed. Toward this end, we will present partial order reduction in the remainder of this section, as well as our new redundancy removal technique in Section V.

For now, we note that, compared to existing test generation tools such as GUITAR [40] and Gazoo [3], the main advantage of Algorithm 1 (baseline) is that it captures all possible event sequences the EFG can produce up to the predefined length. As such, it does not miss useful test sequences.

B. Partial Order Reduction

The idea of partial order reduction originated from explicit-state model checking [49], [41], [19], where the model checker needed to reduce the size of the state space to be searched. In this context, a large number of algorithms were developed, including stubborn set methods, ample set methods, and persistent set methods. For a comprehensive review of these classic methods, refer to Godefroid's book [18]. All these classic methods rely on the same principle, which is first classifying the execution traces into equivalence classes of permutations, and then exploring one representative from each equivalence class. Since all traces from the same equivalence class lead to the same system behavior, covering all equivalence classes is the same as covering all execution traces.

In Fig. 4, for example, the following sequences are considered equivalent: $\{\ldots, \texttt{CB}, \texttt{SA}, \ldots\}$ and $\{\ldots, \texttt{SA}, \texttt{CB}, \ldots\}$. The reason is that `CB` only writes to `convert` and `SA` only reads from `angle`; thus, the execution order of these two events is immaterial. If $\{\ldots, \texttt{CB}, \texttt{SA}, \ldots\}$ has been explored, then $\{\ldots, \texttt{SA}, \texttt{CB}, \ldots\}$ can be skipped.

In Algorithm 1, we add Lines 9 and 12 to show a particular implementation of POR based on the *sleep-set* [18]. Specifi-

Algorithm 1 Baseline test generation procedure with POR.

```
1: Let StateStack S = {s0}, s0.enabled = initially-enabled events, and invoke EXPLORE(S)
2: procedure EXPLORE( S )
3:     let s = S.top()
4:     if (S.size() ≤ MAXLENGTH) then
5:         let s.done = ∅
6:         while ∃event ∈ (s.enabled \ s.done \s.sleep ) do
7:             add event to s.done
8:             let s' = NEXTSTATE(s, event)      // Set s.selEV = event
9:             let s'.sleep = {e ∈ s.sleep | e and event are independent }
10:            S.push(s')
11:            EXPLORE(S)
12:            add event to s.sleep
13:        end while
14:    end if
15:    if (s.selEV = NULL) then
16:        OUTPUTSEQUENCE ( S )          // End trace: ∀s ∈ S, print s.selEV
17:    end if
18:    S.pop()
19: end procedure
```

cally, after an event (e.g., CB in Fig. 4) is explored from a state s (Line 11), it is put into the set $s.sleep$ (Line 12). When we explore another event (e.g., SA) from s, we check if the new event is independent of events in $s.sleep$. If that is the case, we carry the sleep set over to the next state s'. Otherwise, we drop it from $s'.sleep$. At any time in the future, if the procedure EXPLORE attempts to execute a new event (e.g., CB) that already exists in the sleep set (e.g., as in $\{\ldots, \texttt{SA}, \texttt{CB}\}$), we skip the event, because executing the event is guaranteed to reach a previously explored state.

Although POR is widely used [3], [33], [34]), it has a limitation. That is, POR can only identify redundant sequences that are permutations of each other, which implies these sequences have the same set of events; it can never identify redundancy in sequences such as $\{\texttt{SA}, \texttt{SA}, \texttt{OK}\}$ and $\{\texttt{SA}, \texttt{OK}\}$ in Fig. 6 because they have different lengths. Therefore, we need to develop a more powerful reduction technique.

V. THE NEW REDUCTION TECHNIQUE

We first explain the rationale behind our new reduction technique and then present our stateless implementation.

A. The New Algorithm

Algorithm 2 shows our method, which is Algorithm 1 augmented with two modifications at Lines 8 and 17. That is, prior to executing an event (Line 8), we check if the new state s' is a previously explored state by analyzing the events stored in S ($s.selEV \in S$). Similarly, prior to printing an event sequence (Line 17), we check if it can be subsumed by other sequences. Both of these checks are designed to be conservative in nature, meaning if they return *true*, we can safely skip the corresponding states and sequences.

The subroutine REDUNDANTSTATE takes the current state stack S and the next event ev as an input. Recall that states in S are abstract states that do not have concrete values of the variables. Instead, we rely on the event sequence stored in the $selEV$ field of each $s \in S$ to check if the next state has been explored. Thus, our implementation is *stateless*. Similarly, REDUNDANTSEQUENCE checks if the event sequence stored in S can be subsumed by other sequences.

Consider our running example in Fig. 6. The subroutine REDUNDANTSTATE returns *true* when the current sequence in

Algorithm 2 New test generation procedure with reduction.

```
1: Let StateStack S = {s0}, s0.enabled = initially-enabled events, and invoke EXPLORE(S)
2: procedure EXPLORE( S )
3:     let s = S.top()
4:     if (S.size() ≤ MAXLENGTH) then
5:         let s.done = ∅ and s.printed = ∅
6:         while ∃event ∈ (s.enabled \ s.done \s.sleep ) do
7:             add event to s.done
8:             if ¬ REDUNDANTSTATE(S, event) then
9:                 let s' = NEXTSTATE(s, event)      // Set s.selEV = event
10:                let s'.sleep = {e ∈ s.sleep | e and event are independent }
11:                S.push(s')
12:                EXPLORE(S)
13:            end if
14:            add event to s.sleep
15:        end while
16:    end if
17:    if ¬ REDUNDANTSEQUENCE(S, s) then
18:        OUTPUTSEQUENCE (S)          // End trace: ∀s ∈ S, print s.selEV
19:    end if
20:    S.pop()
21: end procedure
22: procedure REDUNDANTSTATE(S, ev)
23:    if NoWrite () ∨ SameWrite () ∨ CovWrite () ∨ GenCovWrite () then
24:        return true
25:    else
26:        return false
27:    end if
28: end procedure
29: procedure REDUNDANTSEQUENCE(S, s)
30:    if (s.selEV ∉ s.printed) ∧ ( IrrelevantTail () ∨ extraSink () ∨ CausalIndep () ) then
31:        return true
32:    else
33:        ∀s ∈ S, add s.selEV to s.printed
34:        return false
35:    end if
36: end procedure
```

S plus the next event, denoted $[S] :: \{event\}$, contains the following sequences:

- $[\texttt{CB}] :: \{\texttt{CB}\} \rightarrow$ state c_1;
- $[\texttt{CB}, \texttt{SL}] :: \{\texttt{CB}\} \rightarrow$ state d_5;
- $[\texttt{CB}, \texttt{SL}] :: \{\texttt{SL}\} \rightarrow$ state d_6;
- $[\texttt{CB}, \texttt{SL}] :: \{\texttt{SA}\} \rightarrow$ state d_7;
- $[\texttt{CB}, \texttt{SL}] :: \{\texttt{OK}\} \rightarrow$ state d_8 (not redundant sequence);
- $[\texttt{CB}] :: \{\texttt{SA}\} \rightarrow$ state c_3;
- $[\texttt{CB}] :: \{\texttt{OK}\} \rightarrow$ state c_4 (not redundant sequence);
- $[\texttt{SL}] :: \{\texttt{CB}\} \rightarrow$ state c_5;
- $[\texttt{SL}] :: \{\texttt{SL}\} \rightarrow$ state c_6;
- $[\texttt{SL}] :: \{\texttt{SA}\} \rightarrow$ state c_7 (not redundant sequence);
- $[\texttt{SL}] :: \{\texttt{OK}\} \rightarrow$ state c_8 (not redundant sequence);
- $[\texttt{SA}] :: \{\texttt{CB}\} \rightarrow$ state c_9.
- $[\texttt{SA}] :: \{\texttt{SL}\} \rightarrow$ state c_{10}.
- $[\texttt{SA}] :: \{\texttt{SA}\} \rightarrow$ state c_{11}.
- $[\texttt{SA}] :: \{\texttt{OK}\} \rightarrow$ state c_{12}.
- $[\texttt{SA}] ::\rightarrow$ state b_3 (not redundant sequence);
- $\{\texttt{OK}\} \rightarrow$ state b_4 (not redundant sequence);

That is, we backtrack as soon as reaching any of the states c_1, d_{5-8}, c_{3-12}, and b_{3-4}, because REDUNDANTSTATE shows they are already explored. In addition, except for six of these sequences, REDUNDANTSEQUENCE proves they are subsumed by some shorter sequences.

Among the six event sequences that are not redundant, $\{\texttt{SA}\} ::\rightarrow b_3$ deserves further explanation since it is the only one that is the prefix of some longer sequences. Furthermore, none of these long sequences (extensions of $\{\texttt{SA}\}$) was printed (because they are redundant sequences themselves). Thus, upon reaching Line 17 of Algorithm 2, we invoke OUTPUTSEQUENCE to print it out.

To enable the printing of partial sequences such as $\{\texttt{SA}\} ::\rightarrow b_3$, we add $s.printed$ to record the set of events

printed at s. Initially, $s.printed$ is empty (Line 5). Every time REDUNDANTSEQUENCE returns *false* (which forces the current sequence to be printed), we add $s.selEV$ to $s.printed$ (Line 33). Thus, only if $(s.selEV \notin s.printed)$ (Line 30), we allow REDUNDANTSEQUENCE to return *true*. Otherwise, the current sequence would have already been printed as part of a longer sequence.

B. Detecting Redundant States

Now, we present the sufficient conditions for detecting already explored states (Line 8 of Algorithm 2). Let

- s_{n-1} and s_n be the last two states in the state stack S,
- ev_{n-1} be the event chosen (and executed) at s_{n-1}, and
- ev be the event considered (but not yet executed) at s_n.

We start with special cases *NoWrite* and *SameWrite*, which are easier to understand, before presenting the general cases.

NoWrite(). The first sufficient condition for $[S] :: \{ev\}$ to result in a redundant state is as follows:

$$(ev_{n-1}.write = \emptyset) \wedge (ev \in s_{n-1}.enabled)$$

Proof sketch: As shown in Fig. 8 (a), since $ev_{n-1}.write = \emptyset$, we know executing ev_{n-1} does not change the state. Thus, $s_n = s_{n-1}$. Furthermore, since $ev \in s_{n-1}.enabled$, the sequence $\{ev_1, \ldots, ev_{n-2}, ev\}$ always exists, and is shorter than $[S] :: \{ev\} = \{ev_1, \ldots, ev_{n-1}, ev\}$. Thus, executing ev from S would not lead to any new program behavior.

SameWrite(). The second sufficient condition for $[S] :: \{ev\}$ to result in a redundant state is as follows:

$$(ev_{n-1}.write \cap ev_{n-1}.read = \emptyset) \wedge (ev = ev_{n-1})$$

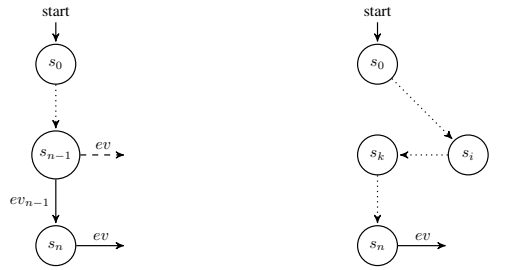
Proof sketch: First, since $ev = ev_{n-1}$, we know the condition $(ev \in s_{n-1}.enabled)$ holds as well. Furthermore, since $ev_{n-1}.read \cap ev_{n-1}.write = \emptyset$, the values read by ev (and hence the values written by ev) do not depend on the values written by ev_{n-1}. In other words, executing ev more than once results in the same state. Thus, executing ev from S would not lead to any new program behavior.

For example, in Fig. 2, $\{\ldots, \texttt{SL}, \texttt{SL}\}$ satisfies this condition. Although `SL` reads from both `convert` and `angle`, the read of `angle` is dominated by its own write to `angle`. Thus, we do not consider `angle` as part of `SL`'s read-variable set. Consequently, `SL` does not causally depend on values written by the previous `SL` .

CovWrite(). This is a generalization of the two previous cases. In this case, the sufficient condition for $S :: \{ev\}$ to result in a redundant state is as follows:

$$\begin{array}{l}(ev_{n-1}.write \subseteq ev.write) \wedge \\ (ev_{n-1}.write \cap ev.read = \emptyset) \wedge \\ (ev \in s_{n-1}.enabled)\end{array}$$

Proof sketch: The first condition means ev overwrites all values written by ev_{n-1}, the second condition means ev_{n-1} does not affect ev via shared variables, and the third condition means ev is enabled at s_{n-1} as well. Therefore, the shorter sequence $\{ev_1, \ldots, ev_{n-2}\} :: \{ev\}$ would lead to the same state as the longer sequence $[S] :: \{ev\} = \{ev_1, \ldots, ev_{n-1}, ev\}$. Thus, we can safely skip the execution of ev from S.



(a) *NoWrite()* (b) *GeneralizedCovWrite()*

Fig. 8. Conditions for s_n to be a redundant state.

GeneralizedCovWrite(). This is a further generalization of the previous case. Let $s_1, \ldots, s_i, \ldots, s_k, \ldots, s_n$ be the entire sequence of states currently in S, where ev_i and ev_k are events selected at s_i and s_k, respectively, and ev is the event selected (but not yet executed) at s_n. The sufficient condition for $[S] :: \{ev\}$ to result in a redundant state is as follows:

$$\begin{array}{ll}\exists 1 \leq i < n \ . & (ev_i.write \subseteq ev.write) \wedge \\ & (ev_i.write \cap \bigcup_{i<k<n} ev_k.read = \emptyset) \wedge \\ & (ev_i.write \cap ev.read = \emptyset) \wedge \\ & (ev_{i+1} \in s_i.enabled)\end{array}$$

Proof sketch: As shown in Fig. 8 (b), the current event ev overwrites all values written by ev_i. Furthermore, ev_i does not affect any of the subsequent events including ev. Furthermore, since $ev_{i+1} \in s_i.enabled$, there is a sequence $\{ev_1, \ldots, ev_{i-1}, ev_{i+1}, \ldots, ev_n, ev\}$ that is shorter and results in the same state as $[S] :: ev$. In this case, executing ev from S will not lead to any new program behavior, because the next state can be reached by some shorter sequence.

C. Eliminating Redundant Sequences

Now, we present our sufficient conditions for detecting redundant sequences (Line 17 of Algorithm 2). These reductions are complementary to POR because they consider a sequence as redundant if it is subsumed by some other sequences of shorter length.

Specifically, in Algorithm 2, prior to generating the event sequence (Line 17), we check if any of the following conditions is satisfied. If the answer is yes, we skip the sequence because the equivalent but shorter sequence would be generated.

IrrelevantTail(). The first sufficient condition for $[S]$ to be a redundant sequence is as follows. Let $s_1, \ldots, s_i, \ldots, s_n$ be the state sequence currently in S, ev_i be the event selected at s_i, and ev_n be the event selected (and executed) at s_n. The sequence $[S]$ is redundant if $\exists 1 \leq i < n$ such that

$$\left(ev_n.read \cap \bigcup_{i \leq k < n} ev_k.write\right) = \emptyset \wedge (ev_n \in s_i.enabled)$$

Proof sketch: When the above condition is satisfied, the last event ev_n (selected and executed at s_n) is guaranteed not to depend on any value written by the preceding events $ev_i, \ldots, ev_{n-1}$. In such case, $[S] = \{ev_1, \ldots, ev_n\}$ can be replaced by the two shorter sequences $\{ev_1, \ldots, ev_{n-1}\}$ and $\{ev_1, \ldots, ev_{i-1}, ev_n\}$, and thus can be skipped.

In our example, $\{\texttt{CB}, \texttt{SA}\}$ has an irrelevant tail and thus can be replaced by the two shorter sequences $\{\texttt{CB}\}$ and $\{\texttt{SA}\}$.

ExtraSink(). Let $s_1, \ldots, s_i, \ldots, s_j, \ldots, s_n$ be the state sequence in S, ev_i and ev_j be the events selected at s_i and s_j, respectively, and ev_n be the event selected (and executed) at s_n. The sequence $[S]$ is redundant if $\exists 1 \leq i < j \leq n$ such that (1) e_i and e_j do not enable or disable any event executed after them, and (2) the following condition is met:

$$\left(e_i.write \cap \bigcup_{i<k\leq n} e_k.read\right) = \emptyset \wedge$$
$$\left(e_j.write \cap \bigcup_{j<k\leq n} e_k.read\right) = \emptyset$$

Proof sketch: The condition means neither e_i nor e_j can affect any event executed after them in S. Furthermore, skipping e_i or e_j does not enable/disable other events. Thus, $[S]$ can be replaced by the shorter sequences $[S] \setminus \{e_i\}$ and $[S] \setminus \{e_j\}$.

For example, in Fig. 6, $\{\texttt{CB}, \ldots, \texttt{SA}, \texttt{OK}\}$ has two extra sinks SA and OK. Therefore, it can be replaced by the two shorter sequences $\{\texttt{CB}, \ldots, \texttt{SA}\}$ and $\{\texttt{CB}, \ldots, \texttt{OK}\}$.

CausalIndependentWrite(). The third sufficient condition is related to causally-independent writes. Let $s_1, \ldots, s_i, \ldots, s_n$ be the state sequence in S, ev_i be the event selected at s_i and ev_n be the event selected (and executed) at s_n. The sequence $[S]$ is redundant if $\exists 1 \leq i < n$ such that (1) ev_i does not enable/disable any event executed after it in S and (2) the following condition is met:

$$(ev_n, ev_i) \notin R_{CD}$$

Proof sketch: First, the above condition means ev_n is not causally dependent on ev_i. In other words, whether ev_i is executed at s_i does not affect the behavior of ev_n. Furthermore, since ev_i does not enable or disable any event executed after it, there exist two shorter sequences $\{ev_1, \ldots, ev_{n-1}\}$ and $\{ev_1, \ldots, ev_{i-1}, ev_{i+1}, \ldots, ev_n\}$ that subsume $[S]$. Thus, the event sequence $[S]$ can be skipped.

D. Computing Causal Dependencies

Whether two events ev_i and ev_j are causally dependent, i.e., $(ev_i, ev_j) \in R_{CD}$, can be decided using a conservative static analysis of their listener functions. The analysis is conservative in that, if it says ev_i is not causally dependent on ev_j, the behavior of ev_i is guaranteed not to be affected by e_j. However, the analysis may not identify all causally independent event pairs due to limitations of static analysis.

We use Soot [48] to implement the static analysis. We mark each Java class member as `className.memberName` and consider all program variables. First, we compute, for each event listener function, the set of read variables and the set of write variables. The difference between our *read* and *write* variable sets and those computed by conventional techniques is that we exclude, from the *read* set, variables that are overwritten before they are read.

Specifically, for each event listener function, we parse the Java byte-code and initialize an empty *write* variable set. For each basic block through which the data flows, we take the union of the in-flow and out-flow. For each basic block where two data flows merge, we take the intersection. When we compute the *read* variable set, if the variables read by a basic block is included in the previously-computed flow set, we ignore them, because they have been overwritten by the method itself. Otherwise we add them to the *read* variable set.

After the aforementioned intra-procedural analysis is completed, we use an inter-procedural program slicer [25] similar to the one used by Gazoo [17] to compute the causal-dependency relation R_{CD}. Essentially, the program slicer recursively adds variables read or written by the listener function as well as functions invoked by the listener function.

As an example, consider the SL event in Fig. 2. Although both `angle` and `convert` are read by the listener function, since `angle` is overwritten before it is read, it is excluded from the *read* set. Thus, $\texttt{SL}.read = \{\texttt{convert}\}$ and $\texttt{SL}.write = \{\texttt{angle}\}$, and therefore $(\texttt{SL}, \texttt{SL}) \notin R_{CD}$.

VI. Experiments

We have implemented our method in a tool named GUICat in which the following components are used: GUITAR [21] for reverse engineering the event flow graph, Soot [44] for conducting static program analysis, and Cobertura [12] for executing test sequences on the GUI application to obtain the coverage report. Our core algorithm for generating test sequences was implemented in 4,000 lines of Java code. For experimental comparison, we implemented the algorithm in such a way that individual reduction techniques can be enabled and disabled. Thus, we were able to compare the performance of the following configurations: (1) our baseline procedure as shown in Algorithm 1, (2) baseline with POR, (3) baseline with POR plus the individual reductions presented in Section V, and (4) all reductions in Algorithm 2 combined. We also downloaded GUITAR [40], [21] and Gazoo [3], [17] and experimentally compared them with our tool on the same benchmark applications.

In both GUITAR and Gazoo, the test sequence generation is model-based. That is, they leverage the same EFG as in our method, but differ in how event sequences are constructed. In our method, the construction starts from the initial states and proceeds systematically, but in GUITAR and Gazoo, the construction may start from any node in the EFG. As such, their initial set of event sequences may not be feasible. To make them feasible—meaning they can be executed by the GUI application—GUITAR and Gazoo have to insert *connecting events* to these sequences. In contrast, our method can directly generate feasible event sequences.

Our experiments were designed to answer two questions:

- Can our new method, which soundly generates test sequences, outperform state-of-the-art GUI testing tools such as GUITAR and Gazoo?
- How effective is our semantic reduction technique and its stateless implementation in identifying and eliminating redundant sequences?

We used 17 benchmark applications written using Java Swing. Their statistics are shown in Table I. Specifically, Columns 1 and 2 show the name of each application and the number of lines of code (LoC), respectively. Note that the LoCs of *regextester* and *hashvcalc* (marked with asterisks) are estimated by decompiling the Java byte-code due to lack

TABLE I
STATISTICS OF THE GUI BENCHMARK APPLICATIONS.

Name	LoC	#Node	#Edge	Description
GazooV0.1	80	5	12	Example application taken from [2].
guess	126	4	16	Example crafted to illustrate scenarios similar to GazooV0.1
GazooV0.2	165	4	16	Example application taken from [3].
ticketseller	367	11	121	GUI application taken from [6]
hashvcalc	*376	17	162	Hash value calculator from sourceforge [24]
workout	526	9	81	Workout generator taken from [6]
payment	665	19	127	Payment form application from the example of squish [45]
regextester	*756	14	144	Java regular expression tester from sourceforge [42]
addressbook	1,267	17	71	Address book application from the example of squish [45]
jnotepad	1,378	45	649	Notepad application from [28]
crosswords	3,594	29	106	Dictionary application [13]
jgp	7,739	71	2,191	GNU plot front-end application from sourceforge [29]
calc	11,940	83	1,634	Student math calculator from sourceforge [9]
ce	14,027	98	1,361	Java class file editor from sourceforge [10]
terpspread	15,231	306	3,079	Java spreadsheet application [47]
rachota	18,852	148	1,347	Time management application from sourceforge and [3]
buddi	28,848	103	927	Financial management application [8]

TABLE II
COMPARISON OF EVENT SEQUENCE REDUCTION.

Name	GUICat (our tool)						GUITAR [21]		Gazoo [3]	
	Baseline		+POR		+AllNew					
	tests	time (s)	tests	time (s)	tests	time (s)	tests	time (s)	tests	time (s)
GazooV0.1	31	2	19	2	3	2	33	1	6	6
guess	64	3	40	3	13	2	64	2	8	5
GazooV0.2	40	2	29	3	6	2	36	1	7	6
ticketseller	1,331	17	418	8	23	3	1,331	16	2	5
hashvcalc	326	6	198	5	20	3	1,657	20	-	6
workout	729	12	246	6	16	2	729	9	-	7
payment	1,009	14	214	5	19	2	1,009	13	-	8
regextester	1,470	24	334	12	25	9	1,470	18	-	33
addressbook	125	4	63	4	20	2	341	6	340	11
jnotepad	8,243	87	1,019	15	158	6	9,916	108	333	13
crosswords	417	9	140	6	30	4	473	7	195	13
jgp	70,335	728	38,189	599	10,613	394	72,675	734	22,610	307
calc	4,179	54	1,183	74	68	28	30,889	327	82	30
ce	21,484	233	5,191	232	755	219	21,538	222	28,172	260
terpspread	15,303	165	4,840	111	3,045	107	40,299*	300*	63,184	1,080
rachota	5,036	67	1,646	39	135	18	22,588*	180*	34,981	1,080
buddi	1,666	32	318	20	60	16	9,021	52	28,008	50
Total	**131,788**	1,459	**54,087**	1,144	**15,009**	819	**214,069**	2,016	**177,928**	2,920

of source code. Columns 3 and 4 show the size of the input EFGs including the number of nodes and the number of edges. Finally, Column 5 provides a brief description of the nature of each application. Together, the benchmark applications have 105,937 lines of Java code. We performed all experiments on a computer with a 3.3 GHz CPU and 8 GB RAM.

A. Comparison of Different Methods

In this experiment, we compared the performance of our tool against GUITAR and Gazoo. Within our own tool, we also compared three configurations: Baseline, with POR, and with the full-blown reduction.

Table II shows the results. Specifically, Column 1 shows the name of each benchmark while the remaining columns are divided into the following groups: the baseline algorithm (denoted *Baseline*), baseline with POR (denoted *+POR*), baseline with POR plus our new reduction (denoted *+AllNew*), and the two existing tools. For each method, we show the number of test sequences generated and the test generation time in seconds. The last row sums up the numbers in each column.

Following Arlt et al. [3] we set the maximum sequence length to 3 for all methods. Note that Gazoo was not able to generate any test sequences for four applications because the tool was hard-wired to produce sequences of dependent events with length 2 or longer, but these four applications do not have sequences that meet the criterion.

TABLE III
EFFECTIVENESS OF DIFFERENT REDUCTION TECHNIQUES.

Name	+POR	Individual Reduction in Our Tool							AllNew
		+NoW	+SameW	+CovW	+GCovW	+IrrTail	+ExtraS	+CauInd	
GazooV0.1	19	19	19	3	10	4	4	8	3
guess	40	40	34	30	38	20	19	22	13
GazooV0.2	29	29	29	14	24	17	12	13	6
ticketseller	418	418	311	281	399	61	47	74	23
hashvcalc	198	145	161	170	194	39	133	47	20
workout	246	140	186	186	240	35	127	51	16
payment	214	116	153	191	211	22	88	52	19
regextester	334	153	235	267	328	44	184	67	25
addressbook	63	40	53	55	63	23	28	28	20
jnotepad	1,019	524	981	972	1,016	181	814	281	158
crosswords	140	69	101	122	138	35	100	59	30
jgp	38,189	31,296	36,868	37,053	38,173	18,318	16,639	15,678	10,613
calc	1,183	928	1,145	968	1,172	498	725	195	68
ce	5,191	3,664	4,902	5,116	5,184	1,801	2,728	1,123	755
terpspreads	4,840	4,498	4,744	4,788	4,837	4,000	3,617	3,204	3,045
rachota	1,646	897	1,417	1,618	1,646	358	997	299	135
buddi	318	109	266	318	318	73	228	110	60
Total	**54,087**	**43,085**	**51,605**	**52,152**	**53,991**	**25,529**	**26,490**	**21,311**	**15,009**

Overall, there is a significant reduction (59%) in the number of test sequences from *Baseline* to *+POR*, and another significant reduction (72%) to *+AllNew*. This means our method is different from and complementary to POR. Furthermore, there are fewer sequences generated by our method (*+AllNew*) than GUITAR and Gazoo, and the reduction is significant (14.3X over GUITAR and 11.9X over Gazoo). The reduction is obtained despite that our method is sound whereas GUITAR and Gazoo may miss important corner cases, as illustrated by our running example in Section II.

The reason why *Baseline* had fewer test sequences than GUITAR and Gazoo was because, as we have mentioned, neither GUITAR nor Gazoo could guarantee their initial set of event sequences were feasible. Thus, they had to insert *connecting events* afterward, which means the final sequences might not be strictly bounded by the MAXLENGTH. There were also no easy fixes that could force them to strictly adhere to the bound. In contrast, the sequences generated by our method were guaranteed to be feasible and within the MAXLENGTH.

The time taken by all test generation methods are more or less the same. Since they all work on the EFG as opposed to executing the actual GUI application, the time is negligible compared to the time taken to execute the test sequence.

B. Comparison of Individual Reduction Techniques

In this experiment, we evaluated the effectiveness of the individual reduction techniques in our method. Table III shows our results, where Column 1 shows the name of each application, Columns 2-10 show the number of test sequences generated by each reduction technique, and the last column shows the number of test sequences generated by all reductions techniques combined. POR was used in conjunction of all the individual reduction techniques.

The results show that each technique is effective compared to the baseline with POR (denoted *+POR*) with improvement ranging from 0.2% to 60% (e.g., computed by $CauInd = 1 - 21311/54087 = 60.60\%$). Furthermore, when combined, they can achieve the largest reduction (72%). This not only means each reduction technique makes its own contribution, but also means they are complementary to each other. For

TABLE IV
COMPARISON OF THE TEST EXECUTION RESULTS.

Name	our tool (new)		GUITAR [40]		Gazoo [3]	
	coverage	tests	coverage	tests	coverage	tests
ce	33%	755	33%	21,538	33%	28,172
terpspreads	56%	3,045	45%	40,299	48%	63,184
rachota	64%	135	62%	22,588	64%	34,981
buddi	36%	60	36%	9,021	36%	28,008

brevity, we do not show the time taken by these individual methods but they are almost the same.

C. Comparison of Test Execution Results

Finally, we compare the test execution. Since running test sequences generated by all methods on all applications takes a long time, we only obtained results on four larger applications. Table IV shows the results, including the name, the percentage of statements covered, and the number of test sequences. The results show all methods achieved a similar coverage. The main difference is in the number of test sequences: it is 3,995 for our method, 93,446 for GUITAR, and 154,345 for Gazoo. For `buddi`, the reduction is 430X: it is 60 sequences for our tool compared to 28K sequences for Gazoo.

D. Threats to Validity

We did not consider external dependencies imposed by remote network communication, database access, or the file IO. Therefore, our method may miss useful event sequences in the presence of these external dependencies. This limitation is shared by the other GUI testing tools as well. We did not consider the diversity of data input either. During our experiments, the data input was generated by GUITAR's replayer using its default setting, to allow a fair comparison of all tools. However, it was also the reason why the testing coverage did not come close to 100%. Same as GUITAR and Gazoo, we focused on only one aspect of GUI testing, which is the diversity of event sequences. To improve further, fuzzing or symbolic execution techniques [23], [31], [22], [22], [11] may be needed to diversify input data; we leave this for future work.

VII. RELATED WORK

GUI is an indispensable component of many software applications. Thus, there has been abundant research on improving the efficiency of GUI testing in various domains, including desktop [60], [57], [11], [56], mobile [27], [38], [1], [26], and web applications [53], [46], [4]. Although techniques proposed in this work were implemented in GUICat [11], which is designed for testing desktop applications, the underlying principle may be applied to other types of GUI applications and event-driven programs in general.

GUI testing is a complex process that requires efficient algorithms and implementation techniques in many different aspects such as static program analysis, dynamic model extraction [37], [54], deterministic replay [20], [58], and test case maintenance [53]. In this work, we focus on event sequence generation only while relying on a number of existing tools such as GUITAR [40], Soot [48], and Cobertura [12] to offer an end-to-end solution. However, there is still room for improvement in these other aspects.

Beside the work mentioned so far, there are other GUI testing techniques [43], [36], [2], [3], [6], [52], [14]. For example, earlier works [43], [52] create models of the software code based on finite state machines, but as pointed out in [5], some of these techniques would not work well when the model does not accurately reflect the actual code. To avoid this problem, Yuan and Memon [57] propose to leverage feedback from the execution of a seed test suite to generate new test cases. Such approach depends on the quality of the seed as well as randomness during test execution.

Our method is related to state-space reduction techniques in explicit-state and symbolic model checking, but with some important differences. In model checking, existing methods are either *model-based* [49], [41], [19], [18], e.g., relying on a state-transition system where values of state variables are available, or *stateless* [15], [55], [50], [32] where the model checker does not maintain states but instead dynamically executes the software. In contrast, our method is a hybrid approach that augments an *abstract model* (the EFG) with dependencies derived from the software *statically*. The EFG is more abstract than the state-transition system because it does not contain values of the program variables.

Test sequence reduction has been studied in event-driven programs [3], [2], [7], [36] to reduce the test execution cost. In this context, partial order reduction (POR) [18], [15], [33], [34] serves as a foundational technique for removing redundancy. However, as shown in Section II as well as the experiments, although POR is effective in identifying redundancy among sequences of the same length, it misses other redundant sequences. In comparison, our method is more effective since it also exploits redundancy among sequences of different lengths.

Beyond test sequence generation, an important problem is diversifying the input data. Several recent works have focused on this problem, e.g., by using model checking [35], [39], [46] and symbolic execution [11], [1], [27], [38]. However, scalability remains a problem and thus there is still room for improvement. We will consider it for future work.

VIII. CONCLUSIONS

We have presented a GUI testing framework for efficiently generating event sequences while avoiding the redundant sequences. Our technique leverages both model-driven test generation (e.g., the EFG) and static analysis of the actual software (e.g., the Java bytecode). It goes beyond partial order reduction by identifying redundancy not only among event sequences of the same length but also among sequences of different lengths. Our experiments on Java Swing applications show the new method significantly outperforms state-of-the-art GUI testing tools and the average reduction in the number of test sequences is more than 10X. For future work, we plan to develop methods for diversifying input data to further improve the testing coverage.

IX. ACKNOWLEDGMENTS

This material is based upon research supported in part by the U.S. National Science Foundation under grants DGE-1522883, CCF-1500365, and CCF-1149454, as well as the U.S. Office of Naval Research under award number N00014-17-1-2896.

REFERENCES

[1] Saswat Anand, Mayur Naik, Mary Jean Harrold, and Hongseok Yang. Automated concolic testing of Smartphone apps. In *ACM SIGSOFT Symposium on Foundations of Software Engineering*, pages 59:1–12, 2012.

[2] Stephan Arlt, Andreas Podelski, Cristiano Bertolini, Martin Schäf, Ishan Banerjee, and Atif M. Memon. Lightweight static analysis for GUI testing. In *International Symposium on Software Reliability Engineering*, 2012.

[3] Stephan Arlt, Andreas Podelski, and Martin Wehrle. Reducing GUI test suites via program slicing. In *International Symposium on Software Testing and Analysis*, 2014.

[4] Shay Artzi, Julian Dolby, Simon Holm Jensen, Anders Moller, and Frank Tip. A framework for automated testing of JavaScript web applications. In *International Conference on Software Engineering*, pages 571–580, 2011.

[5] Gigon Bae, Gregg Rothermel, and Doo-Hwan Bae. On the relative strengths of model-based and dynamic event extraction-based GUI testing techniques: An empirical study. In *International Symposium on Software Reliability Engineering*, pages 181–190, 2012.

[6] Barad. A GUI testing framework based on symbolic execution. http://users.ece.utexas.edu/~perry/work/papers/080521-SG-barad.pdf.

[7] Fevzi Belli. Finite state testing and analysis of graphical user interfaces. In *International Symposium on Software Reliability Engineering*, pages 34–43, 2001.

[8] Buddi. http://buddi.digitalcave.ca/.

[9] Calc. https://sourceforge.net/projects/formcalc/.

[10] Ce. http://classeditor.sourceforge.net/.

[11] Lin Cheng, Jialiang Chang, Zijiang Yang, and Chao Wang. GUICat: GUI testing as a service. In *IEEE/ACM International Conference On Automated Software Engineering*, pages 858–863, 2016.

[12] Cobertura. http://cobertura.github.io/cobertura/.

[13] Crosswords. http://www.cs.umd.edu/~atif/Benchmarks/UMD2008a.html.

[14] Pranavadatta Devaki, Suresh Thummalapenta, Nimit Singhania, and Saurabh Sinha. Efficient and flexible GUI test execution via test merging. In *International Symposium on Software Testing and Analysis*, pages 34–44, 2013.

[15] Cormac Flanagan and Patrice Godefroid. Dynamic partial-order reduction for model checking software. In *ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 110–121, 2005.

[16] Malay K. Ganai, Nipun Arora, Chao Wang, Aarti Gupta, and Gogul Balakrishnan. BEST: A symbolic testing tool for predicting multi-threaded program failures. In *IEEE/ACM International Conference On Automated Software Engineering*, 2011.

[17] Gazoo. https://swt.informatik.uni-freiburg.de/tool/gazoo.

[18] Patrice Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems - An Approach to the State-Explosion Problem*. Springer, 1996.

[19] Patrice Godefroid and Pierre Wolper. A partial approach to model checking. *Inf. Comput.*, 110(2):305–326, 1994.

[20] Lorenzo Gomez, Iulian Neamtiu, Tanzirul Azim, and Todd Millstein. Reran: Timing-and touch-sensitive record and replay for Android. In *International Conference on Software Engineering*, pages 72–81, 2013.

[21] GUITAR. https://sourceforge.net/projects/guitar/.

[22] Shengjian Guo, Markus Kusano, Chao Wang, Zijiang Yang, and Aarti Gupta. Assertion guided symbolic execution of multithreaded programs. In *ACM SIGSOFT Symposium on Foundations of Software Engineering*, pages 854–865, 2015.

[23] William G.J. Halfond and Alessandro Orso. Improving test case generation for Web applications using automated interface discovery. In *ACM SIGSOFT Symposium on Foundations of Software Engineering*, September 2007.

[24] Hashvcalc. https://sourceforge.net/projects/hash-value-tester-gui/.

[25] Susan Horwitz, Thomas W. Reps, and David Binkley. Interprocedural slicing using dependence graphs. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 35–46, 1988.

[26] Chun-Hung Hsiao, Jie Yu, Satish Narayanasamy, Ziyun Kong, Cristiano L Pereira, Gilles A Pokam, Peter M Chen, and Jason Flinn. Race detection for event-driven mobile applications. *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 326–336, 2014.

[27] Casper S Jensen, Mukul R Prasad, and Anders Møller. Automated testing with targeted event sequence generation. In *International Symposium on Software Testing and Analysis*, pages 67–77, 2013.

[28] jNotepad. http://www.planet-source-code.com/vb/scripts/ShowCode.asp?txtCodeId=3363&lngWId=2.

[29] JPG. http://jgp.sourceforge.net/.

[30] Vineet Kahlon, Chao Wang, and Aarti Gupta. Monotonic partial order reduction: An optimal symbolic partial order reduction technique. In *International Conference on Computer Aided Verification*, pages 398–413, 2009.

[31] Sudipta Kundu, Malay K. Ganai, and Chao Wang. CONTESSA: Concurrency testing augmented with symbolic analysis. In *International Conference on Computer Aided Verification*, pages 127–131, 2010.

[32] Markus Kusano and Chao Wang. Assertion guided abstraction: a cooperative optimization for dynamic partial order reduction. In *IEEE/ACM International Conference On Automated Software Engineering*, pages 175–186, 2014.

[33] Pallavi Maiya, Rahul Gupta, Aditya Kanade, and Rupak Majumdar. Partial order reduction for event-driven multi-threaded programs. In *International Conference on Tools and Algorithms for Construction and Analysis of Systems*, pages 680–697, 2016.

[34] Pallavi Maiya and Aditya Kanade. Efficient computation of happens-before relation for event-driven programs. In *International Symposium on Software Testing and Analysis*, pages 102–112, 2017.

[35] Peter Mehlitz, Oksana Tkachuk, and Mateusz Ujma. JPF-AWT: Model checking GUI applications. In *IEEE/ACM International Conference On Automated Software Engineering*, pages 584–587, 2011.

[36] Atif M Memon. An event-flow model of GUI-based applications for testing. *Software Testing Verification and Reliability*, 17(3):137–158, 2007.

[37] Atif M. Memon, Ishan Banerjee, and Adithya Nagarajan. GUI Ripping: Reverse engineering of Graphical User Interfaces for testing. In *Proceedings of The 10th Working Conference on Reverse Engineering*, November 2003.

[38] Nariman Mirzaei, Joshua Garcia, Hamid Bagheri, Alireza Sadeghi, and Sam Malek. Reducing combinatorics in GUI testing of Android applications. In *International Conference on Software Engineering*, pages 559–570, 2016.

[39] Nariman Mirzaei, Sam Malek, Corina S Păsăreanu, Naeem Esfahani, and Riyadh Mahmood. Testing Android apps through symbolic execution. *ACM SIGSOFT Software Engineering Notes*, 37(6):1–5, 2012.

[40] Bao N Nguyen, Bryan Robbins, Ishan Banerjee, and Atif Memon. GUITAR: an innovative tool for automated testing of GUI-driven software. *Automated Software Engineering*, 21(1):65–105, 2014.

[41] Doron A. Peled. All from one, one for all: on model checking using representatives. In *International Conference on Computer Aided Verification*, pages 409–423, 1993.

[42] Regextester. https://sourceforge.net/projects/javaregextestgui/.

[43] Richard K Shehady and Daniel P Siewiorek. A method to automate user interface testing using variable finite state machines. In *International Symposium on Fault-Tolerant Computing*, pages 80–88, 1997.

[44] Soot. https://github.com/Sable/soot.

[45] Squish. https://www.froglogic.com/squish/.

[46] Chungha Sung, Markus Kusano, Nishant Sinha, and Chao Wang. Static DOM event dependency analysis for testing web applications. In *ACM SIGSOFT Symposium on Foundations of Software Engineering*, pages 447–459, 2016.

[47] Terpspreadsheet. http://www.cs.umd.edu/~atif/Benchmarks/UMD2007a.html.

[48] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. Soot-a Java bytecode optimization framework. In *Conference of the Centre for Advanced Studies on Collaborative research*, 1999.

[49] Antti Valmari. A stubborn attack on state explosion. In *International Workshop on Computer Aided Verification*, pages 156–165, 1990.

[50] Chao Wang, Mahmoud Said, and Aarti Gupta. Coverage guided systematic concurrency testing. In *International Conference on Software Engineering*, pages 221–230, 2011.

[51] Chao Wang, Zijiang Yang, Vineet Kahlon, and Aarti Gupta. Peephole partial order reduction. In *International Conference on Tools and Algorithms for Construction and Analysis of Systems*, pages 382–396, 2008.

[52] Lee White and Husain Almezen. Generating test cases for GUI responsibilities using complete interaction sequences. In *International Symposium on Software Reliability Engineering*, pages 110–121, 2000.

[53] Rahulkrishna Yandrapally, Giriprasad Sridhara, and Saurabh Sinha. Automated modularization of GUI test cases. In *International Conference on Software Engineering*, pages 44–54, 2015.

[54] Wei Yang, Mukul R Prasad, and Tao Xie. A grey-box approach for automated gui-model generation of mobile applications. In *International*

Conference on Fundamental Approaches to Software Engineering, pages 250–265, 2013.

[55] Yu Yang, Xiaofang Chen, Ganesh Gopalakrishnan, and Chao Wang. Automatic discovery of transition symmetry in multithreaded programs using dynamic analysis. In *International SPIN workshop on Model Checking Software*, pages 279–295, 2009.

[56] Xun Yuan and Atif Memon. Iterative execution-feedback model-directed GUI testing. *Information and Software Technology*, 52(5):559–575, 2010.

[57] Xun Yuan and Atif M Memon. Generating event sequence-based test cases using GUI runtime state feedback. *IEEE Transactions on Software Engineering*, 36(1):81–95, 2010.

[58] Lu Zhang and Chao Wang. RClassify: classifying race conditions in web applications via deterministic replay. In *International Conference on Software Engineering*, pages 278–288, 2017.

[59] Naling Zhang, Markus Kusano, and Chao Wang. Dynamic partial order reduction for relaxed memory models. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 250–259, 2015.

[60] Sai Zhang, Hao Lü, and Michael D Ernst. Finding errors in multithreaded GUI applications. In *International Symposium on Software Testing and Analysis*, pages 243–253, 2012.