Reducing GUI Test Suites via Program Slicing

Stephan Arlt
Université du Luxembourg
Luxembourg City, Luxembourg
stephan.arlt@uni.lu

Andreas Podelski
Universität Freiburg
Freiburg, Germany
podelski@cs.uni-freiburg.de

Martin Wehrle
Universität Basel
Basel, Switzerland
martin.wehrle@unibas.ch

ABSTRACT

A crucial problem in GUI testing is the identification of accurate event sequences that encode corresponding user interactions with the GUI. Ultimately, event sequences should be both feasible (i. e., executable on the GUI) and relevant (i. e., cover as much of the code as possible). So far, most work on GUI testing focused on approaches to generate feasible event sequences. In addition, based on event dependency analyses, a recently proposed static analysis approach systematically aims at selecting both relevant and feasible event sequences. However, statically analyzing event dependencies can cause the generation of a huge number of event sequences, leading to unmanageable GUI test suites that are not executable within reasonable time.

In this paper we propose a refined static analysis approach based on program slicing. On the theoretical side, our approach identifies and eliminates *redundant* event sequences in GUI test suites. Redundant event sequences have the property that they are guaranteed to not affect the test effectiveness. On the practical side, we have implemented a slicing-based test suite reduction algorithm that approximatively identifies redundant event sequences. Our experiments on six open source GUI applications show that our reduction algorithm significantly reduces the size of GUI test suites. As a result, the overall execution time could significantly be reduced without losing test effectiveness.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging

General Terms

Algorithms, Experimentation

Keywords

GUI Testing, Black-box Testing, Test Automation;

ISSTA '14, July 21–25, 2014, San Jose, CA, USA

1. INTRODUCTION

Test case generation for graphical user interfaces (GUIs) is an active research area [3, 8, 18, 31, 34]. Test cases are represented as sequences of *events* that encode corresponding user interactions with the GUI. Test cases should be both feasible (i. e., the event sequence should be executable on the GUI), and relevant in the sense that as much of the code as possible is covered. In this context, a main challenge is the potentially unbounded space of possible user interactions, and hence, the potentially unbounded space of possible event sequences.

Recent approaches to tackle this challenge can be classified as *iterative* and *non-iterative* approaches. Iterative approaches (e. g., [18, 31]) generate test cases on-the-fly, and test cases can be executed after their generation. In order to keep the approach practical, the generation and execution time is usually limited with a timeout. Complementary to iterative approaches, non-iterative approaches (e. g., [3, 8, 34]) generate the whole suite of test cases before their actual execution, which has the advantage that a complete set of test cases (e.g., all event sequences of a specific length) is generated. In this context, most of the work concentrated on black-box approaches is to obtain feasible test cases. For example, Belli [8] proposed the notion of Event Sequence Graphs (ESGs), and Memon [34] proposed the notion of Event Flow Graphs (EFGs) to approximate the (black-box) behavior of the GUI. In addition to black-box approaches, white-box approaches have been proposed for non-iterative test case generation (e. g., [12, 42]) to select relevant test cases based on, e. g., symbolic execution.

A recently proposed approach [3] aims at combining the best of these two worlds by combining black-box and white-box techniques to identify both feasible and relevant sequences. In a first step, the white-box part selects "skeletons" of event sequences based on pairs of events that are in a def-use relationship. Such pairs of events are represented by an event dependency graph (EDG) that is computed statically. In a second step, the black-box part "fills" this skeleton with events such that the overall sequence becomes feasible. Roughly speaking, this approach combines what one "should test" (events that depend on each other identified with the EDG) with what one "can test" (events that are feasible identified with the EFG). While this approach has shown promising performance compared to pure black-box testing, it only considers *pairs* of def-use events for the first (i. e., for the "white-box") step. However, longer event sequences are often useful to detect more complex bugs as, e. g., demonstrated by Xie et al. [48] and Assi et al. [5]. Furthermore, the

number of event sequences that result from pairs of def-use events can nevertheless be huge such that resulting GUI test suites cannot be executed in reasonable time.

In this paper we provide a generalization of this approach. The generalization is not restricted to pairs, but can handle an arbitrary number of dependent events while guaranteeing that all relevant event sequences are generated based on these events. To make this generalization scalable, we propose a refined static analysis approach based on a variant of *program slicing* [19, 45] to reduce the resulting number of test cases. On the theoretical side, our approach identifies and eliminates *redundant* event sequences in GUI test suites. Redundant event sequences have the property that they are guaranteed to not affect the test effectiveness. On the practical side, we have implemented a slicing-based test suite reduction algorithm that approximatively identifies redundant event sequences.

Our experiments on a number of real-world open source GUI applications confirm its practical potential. In particular, our experiments demonstrate that redundant event sequences occur in a huge number of various GUI applications. Moreover, the reduced number of generated test cases can strongly reduce the overall execution time of a GUI test suite without losing test effectiveness.

The remainder of the paper is organized as follows. Next, we motivate the applicability of our approach using an example GUI application. Section 3 formally introduces the concept of redundant event sequences, and Section 4 presents its actual implementation on a technical level based on program slicing. We evaluate the new approach in Section 5, which is followed by a discussion of its results in Section 5.3. Finally, we conclude the paper and sketch future work.

2. MOTIVATION

As outlined in the introduction, the recently proposed static analysis approach for GUI testing [3] has shown its potential on several GUI applications. However, in practice, longer dependent event sequences than those handled by this approach are often desired [5, 48]. Considering longer event sequences in turn leads to an unmanageable number of overall test cases. In the following, we provide a simple motivating example, which is inspired by a real-world GUI application to show these problems in more detail, and to outline our approach to reduce the resulting number of test cases.

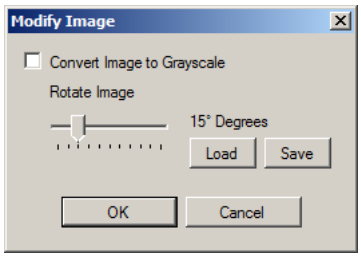


Figure 1: An example GUI inspired by *TerpPaint*.

Consider the example GUI application in Figure 1, which is inspired by the *TerpPaint* application [32] (see Section 5 for more details). The window offers a user to modify a recently opened image. To modify the image, the user may click the checkbox in order to convert the image to grayscale, and may rotate the image by choosing an angle from the slider control. Furthermore, an existing angle can be loaded from the user settings (using the button *Load*), and a new angle can be saved as a user setting (using the button *Save*). The button *OK* performs the modification to the image and closes the window; the button *Cancel* closes the window without any modifications to the image.

```
1  class ModifyImageWindow extends JFrame {
2
3    boolean convert = false;
4    int angle = 0;
5
6    void onCheckBox() {
7      int cbValue = checkBox.getValue();
8      convert = (1 == cbValue)
9        ? true : false;
10   }
11
12   void onSlider() {
13     int sliderValue = slider.getValue();
14     angle = sliderValue;
15     print(convert, angle);
16   }
17
18   void onSave() {
19     UserSettings.RotationAngle = angle;
20   }
21
22   void onOK() {
23     if ( convert ) {
24       image.convertToGrayscale();
25       image = null;
26     }
27     if ( angle > 0 ) {
28       // BUG: crashes if image was
29       // converted to grayscale
30       image.rotate(angle);
31     }
32   }
33 }
```

Figure 2: The event handlers extracted from the example GUI. The class `ModifyImageWindow` defines the event handlers `onCheckBox`, `onSlider`, `onSave`, and `onOK`.

Figure 2 shows a snippet of the code that describes the GUI application. The application contains *event handlers* that define the behavior of the GUI in case a corresponding *event* (i. e., interaction with the user) occurs. We will use the terms event and event handler interchangeably when the meaning is clear from the context. In this example, there are the four event handlers `onCheckBox`, `onSlider`, `onSave` and `onOK`. `onCheckBox` reads the current value from the checkbox and assigns its value to the field `convert`. `onSlider` reads the current angle from the slider control and assigns the value to the field `angle`. It also prints the current values to a log. `onSave` saves the current angle as a user setting. `onOK` converts the image to grayscale and resets the image object; furthermore it rotates the image by the given angle.

As indicated in Figure 2 (line 28-30), the GUI application contains a bug that can occur if the event handler `onOK` is executed after the event handlers `onCheckBox` and `onSlider`: If `onOK` is executed, and the field `convert` is true (set by `onCheckBox`), and also the field `angle` is greater than zero (set by `onSlider`), then the image object is set to null, causing a `NullPointerException` in line 30.

The recently proposed static analysis approach [3] does not reveal this bug, because it generates test cases as follows.

For the given GUI application, an *event dependency graph* (EDG) is computed that reflects the def-use dependencies of the events. The vertices of the EDG are defined as the events of the GUI application, and there is an edge between two events if these events are in def-use relationship. For our concrete example, this graph is depicted in Figure 3.

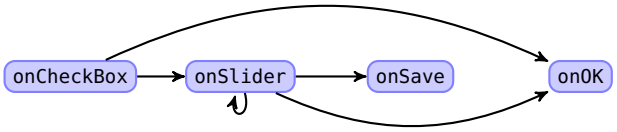


Figure 3: Event Dependency Graph (EDG) for the example GUI. Each edge expresses a def-use (a write/read) dependency: For example, the event `onCheckBox` defines (writes) the field `convert`, which is used (read) in the event `onSlider` and in the event `onOK`.

Based on the EDG, for all pairs of events e and e' such that there is an edge between e and e', a test case "skeleton" that contains e and e' is computed. This sequence is considered relevant, because the events are dependent. In our example, the following five event sequences are generated.

$s_1 = \langle$ onCheckBox, onOK $\rangle$
$s_2 = \langle$ onCheckBox, onSlider $\rangle$
$s_3 = \langle$ onSlider, onSave $\rangle$
$s_4 = \langle$ onSlider, onSlider $\rangle$
$s_5 = \langle$ onSlider, onOK $\rangle$

Since such skeleton sequences are abstract in the sense that they are not necessarily executable on the GUI, a black-box model of the GUI is finally applied (e. g., the approach presented in Memon [34]) in order to transform the event dependency sequence in an executable test sequence. (In our example, all sequences are executable, so there is nothing more to do). At this point, the important observation is that the bug in line 30 is *not* revealed by this approach, as *three* events (`onOK`, `onCheckBox` and `onSlider`) are needed to reveal the bug. Hence, increasing the length from $n = 2$ to $n = 3$ would clearly reveal the bug. However, as the potential number of resulting abstract event sequences becomes huge, techniques to effectively reduce the number of sequences are required. In the following, we outline our approach to identify *redundant* event sequences that can be removed while still obtaining the same code coverage.

2.1 Examples for Redundancy

Assume one would like to test the event `onOK` of the example GUI in Figure 1. Furthermore, assume that the following two event sequences s_1 and s_2 have been generated in order to achieve this.

$s_1 = \langle$ onCheckBox, onSlider, onOK $\rangle$
$s_2 = \langle$ onSlider, onCheckBox, onOK $\rangle$

Both s_1 and s_2 reveal the bug in line 30. In fact, we observe that these sequences are equivalent in the sense that the execution ordering of `onCheckBox` and `onSlider` does not matter for the execution of `onOK` (even though there is an edge in the EDG from `onCheckBox` to `onSlider`). Hence, one of these sequences is *redundant* and can be ignored. This is essentially a simple variant of *partial order reduction* [14, 16] applied to GUI testing. We will make this precise in the next section.

As a further example of redundant event sequences, assume one would like to test the event `onSave` of the example GUI. Further assume that the following event sequence s_3 has been generated in order to achieve this.

$s_3 = \langle$ onCheckBox, onSlider, onSave $\rangle$

Although there is an edge in the EDG from `onCheckBox` to `onSlider`, and from `onSlider` to `onSave`, there is no "causal" data-flow from the first to the third of these events. This is because the variable that causes the data-flow from `onCheckBox` to `onSlider` (i. e., the variable `convert` is written by `onCheckBox` and read by `onSlider`) is different from the variable that causes the data-flow from `onSlider` to `onSave` (i. e., the variable `angle`). Hence, the global effect of s_3 after executing `onSave` is completely independent of the `onCheckBox` event, and it suffices to consider the shorter sequence $\langle$ onSlider, onSave $\rangle$ instead of s_3 to test the `onSave` event. In this sense, s_3 is redundant. (As a side remark, the variables `cbValue` and `sliderValue` are local and can hence be ignored for these considerations). Informally speaking, this problem of "pseudo-dependency" arises because the EDG is computed statically and syntactically, without a deeper analysis of the actual causal data-flow [45]. In the next sections we propose approaches to systematically identify such redundant sequences.

3. REDUNDANT EVENT SEQUENCES

In this section we first introduce a generalization of the test case generation algorithm of Arlt et al. [3]. Afterwards, we propose two approaches to identify sufficient criteria of *redundant* test cases when applied in the context of our generalized algorithm.

3.1 Test Case Generation Algorithm

In this section we formalize the ideas of the previous motivation section, and provide an algorithm for test case generation as a result. To introduce our approach, we identify a GUI application through its corresponding finite set of events V. Let us start our considerations with a short rehash of the definition of *event dependency graphs* [3].

Definition 1 (Event Dependency Graph (EDG)). *The* event dependency graph *for a finite set of events V is defined as the directed graph $EDG = (V, E)$ with the set of vertices V, such that there is an edge $(e, e') \in E$ iff there is a def-use relationship between e and e' (i. e., there is a variable that is defined by e and used by e').*

For a given GUI application, the event dependency graph reflects the pairwise def-use relationships of its events. EDGs reflect an important concept in identifying "relevant" test cases, i. e., test cases that "should be tested" [3]. In the following, we formalize this intuition based on the definition of *relevant* EDG-sequences.

Definition 2 (Relevant Sequence). *Let V be a finite set of events, and let $n \in \mathbb{N}$ be a natural number. Then $\langle e_1, \ldots, e_n \rangle$ is a* relevant sequence *of length n iff for all $i \in \{1, \ldots, n-1\}$, there is a $j \in \{i+1, \ldots, n\}$ and an edge from e_i to e_j in the EDG. The set $EDG(n)$ is defined as the set that contains the relevant sequences of length n.*

Informally speaking, for $n \in \mathbb{N}$, the set $EDG(n)$ contains exactly those event sequences that are causally linked in the sense that for all events e in the sequence, there is at least one event e' that occurs later in the sequence such that e and e' are in def-use relationship (i. e., events occurring later in the sequence can be influenced by events that occur earlier). This definition of relevant sequences actually describes the heart of the generalization of the test case generation algorithm by Arlt et al. [3]. More precisely, for a given $n \in \mathbb{N}$, test cases are generated according to the following procedure *GenerateRelevantTestCases*:

1. For all $i \in \{1, \ldots, n\}$, compute the set $EDG(i)$ that contains all relevant EDG-sequences of length i.
2. For all sequences $s \in EDG(i)$: If s is not executable, use a black-box model of the GUI (i. e., [3, 34]) to enhance s such that it becomes executable.

At this point, we observe that the recently proposed static analysis approach [3] is a special case of this algorithm for $n = 2$. It generates all relevant test cases of length $\leq n$. However, generating all of these test cases results in a huge number that exceeds the number that can be handled in reasonable resource limits even for small n. To tackle this problem to become scalable in practice, we propose two approaches to identify redundant event sequences.

3.2 Partial Order (PO) Redundancy

Partial order reduction [14, 16] is a well-established approach to tackle the state explosion in the area of model checking. Most of the techniques used for partial order reduction are state-dependent, which means that the actual pruning decisions depend on the currently encountered state.

In this section we apply a state-independent technique to identify a class of redundant event sequences for GUI testing. This technique is based on the simple observation that two events that are *independent* in the sense that they can be applied in both possible orderings with the same global effect need not be considered in both, but only in one of these orderings. This idea has been investigated under the name *commutativity pruning* in the area of Artificial Intelligence [23]. In the following, we formalize this intuition.

Definition 3 (Partial Order Redundancy). *Let V be a finite set of events, let $<$ be a total ordering on V. Let $n \in \mathbb{N}$ and $s = \langle e_1, \ldots, e_n \rangle \in EDG(n)$. If*

1. *the values of the variables that are read by e_n are the same after executing $e_1, \ldots, e_{n-1}$ in all possible orderings, and*
2. *the first $n-1$ events do not occur in the "correct" ordering according to the ordering $<$ (i. e., there are events e and e' with $\{e, e'\} \subseteq \{e_1, \ldots, e_{n-1}\}$ such that e' occurs before e in s, but $e < e'$)*

then s is Partial Order redundant (PO-redundant) *with respect to the ordering $<$.*

Informally speaking, an event sequence $s = \langle e_1, \ldots, e_n \rangle \in EDG(n)$ can be PO-redundant if the execution ordering of the first $n-1$ does not matter for the final event e_n. In such cases, it suffices to consider only one out of $(n-1)!$ possible event sequences. Definition 3 formalizes this idea based on the ordering $<$ by implicitly declaring $(n-1)! - 1$ sequences as PO-redundant, and retaining one representative sequence as relevant, i. e., as not PO-redundant. (We will describe how to actually instantiate and implement this definition on a technical level in the next section).

For now, to get a better idea of PO redundancy, consider again the first example of the motivation section restricted to the events $V = \{\texttt{onCheckBox}, \texttt{onSlider}, \texttt{onOK}\}$. Suppose we choose the ordering $\texttt{onCheckBox} < \texttt{onSlider} < \texttt{onOK}$. According to the ordering $<$, and because `onCheckBox` and `onSlider` can be executed in both possible orderings to test `onOK`, the sequence $\langle$ `onSlider`, `onCheckBox`, `onOK` $\rangle$ is PO-redundant. Generally, we observe that PO-redundant event sequences have the property that the test case generation algorithm given in the last section can ignore such sequences without reducing code coverage. More formally, for a finite set of events V and for a total ordering $<$ on V, the sets of generated test cases with algorithm *GenerateRelevantTestCases* based on the event sequences

$$EDG(n) \text{ and } EDG(n) \setminus \{s \mid s \text{ is PO-redundant w.r.t. } <\}$$

are equivalent for all $n \in \mathbb{N}$.

3.3 Causal Variable (CV) Redundancy

To introduce our second technique to identify redundant event sequences (and hence, to reduce the number of test cases), let us first re-consider the second example of the motivation section. On a more formal level, the example consists of the event sequence $s = \langle e_1, e_2, e_3 \rangle \in EDG(3)$. We observe that s is "chain-shaped" in the sense that there is an edge in the EDG between e_1 and e_2, there is an edge between e_2 and e_3, and there are no further edges. Recall that s is executable.

In this example, we observe that the variables causing the def-use relationship of e_1 and e_2 are disjoint from the variables causing the def-use relationship of e_2 and e_3. Hence, despite the edge from e_1 to e_2, the effect of executing e_3 is independent of the effect of e_1.

Therefore, s does not need to be considered, because it suffices to consider the shorter sequences $s' = \langle e_2, e_3 \rangle$ and $s'' = \langle e_1, e_3 \rangle$ to test e_3. To formalize the idea of this reduction technique, we need to talk about the variables that cause def-use relationships.

Definition 4 (Labels of Edges). *Let V be a finite set of events, let $e, e' \in V$. The set of associated labels $labels(e, e')$ to e and e' is defined as the set of variables that are defined by e and used by e'.*

Note that the above definition can be considered as the implicit labeling of the edges in the EDG, because edges are represented as pairs of events. Based on this definition, we formalize the idea of redundant sequences discussed above.

Definition 5 (Causal Variable Redundancy). *Let V be a finite set of events, and let $EDG = (V, E)$ be the event dependency graph (with vertices V and edges E) for V. Furthermore, let $n \in \mathbb{N}$, and let $s = \langle e_1, \ldots, e_n \rangle \in EDG(n)$ such that there are edges $(e_i, e_{i+1}) \in E$ for $i \in \{1, \ldots, n-1\}$ in the EDG, and there are no further edges between events in s. If there exist events e, e', e'' such that $\{e, e', e''\} \subseteq \{e_1, \ldots, e_n\}$ such that*

1. *$(e, e') \in E$, $(e', e'') \in E$, and*

2. $labels(e, e') \cap labels(e', e'') = \emptyset$,

then s is Causal Variable redundant (CV-redundant)*, i. e., redundant with respect to causal variable analysis.*

In the above example, the sequence s is CV-redundant, because $labels(e_1, e_2) \cap labels(e_2, e_3) = \emptyset$. Similarly to PO-redundant sequences, CV-redundant sequences need not be considered w.r.t. code coverage. More formally, for a finite set of events V and the event dependency graph $EDG = (V, E)$ for V, the sets of generated test cases with algorithm *GenerateRelevantTestCases* based on the event sequences

$$EDG(n) \text{ and } EDG(n) \setminus \{s \mid s \text{ is CV-redundant}\}$$

are equivalent for all $n \in \mathbb{N}$.

4. SLICING-BASED IMPLEMENTATION

In the last section we have provided the theoretical background to identify redundant event sequences. In this section we put these theoretical notions of redundancy into practice. A central concept of our implementation is the concept of *program slicing* in the sense of Gupta et al. [19]. In this context, a program slice is based on direct and indirect def-use associations of variables in the source code. We particularly address the questions of how to compute such program slices, and how to implement the two notions of redundancy. The overall implementation is depicted in Figure 4 and contains the three main components *Body Transformer*, *Program Slicer*, and *Sequence Generator*. In the following, we provide a more detailed description.

4.1 Soot

In a first step, the *Soot* component performs a source-to-source transformation of the input GUI application to an appropriate intermediate format. In particular, the intermediate format is supposed to appropriately reflect the required information about defs (definitions) and uses for the further analysis (e. g., the bytecode analysis tool ASM [4] is not appropriate for our purpose since we would have to build an interpreter, which analyzes the stack of bytecode instructions [30]). The defs and uses are necessary in order to compute the program slices in a later step.

To compute such an intermediate format, our implementation applies the tool *Soot* [28]. Soot takes the bytecode [30] of the GUI application as input, and transforms it to *Jimple* code [44]. Jimple code is three-address code, which is an intermediate representation of bytecode that simplifies the program analysis. Most importantly, Soot provides an API to extract the defs (definitions) and uses of each statement in a method. A statement is called *Unit* in Jimple code.

4.2 Body Transformer

The body transformer takes as input the Jimple code, and returns the set of defs and uses of fields and (local) variables of the GUI application. This information is later used in the program slicer in order to create *backward slices* [45] of fields. Roughly speaking, a backward slice of a *def* expresses which previous *uses* in the Jimple code (and thus, in the Java code) affect the definition of the field. To generate the necessary information about defs and uses, the body transformer analyzes the units of the Jimple code (which are comparable to statements in Java code). For each unit u, the defs and uses are extracted and maintained in a mapping $Defs(u)$ and $Uses(u)$, respectively.

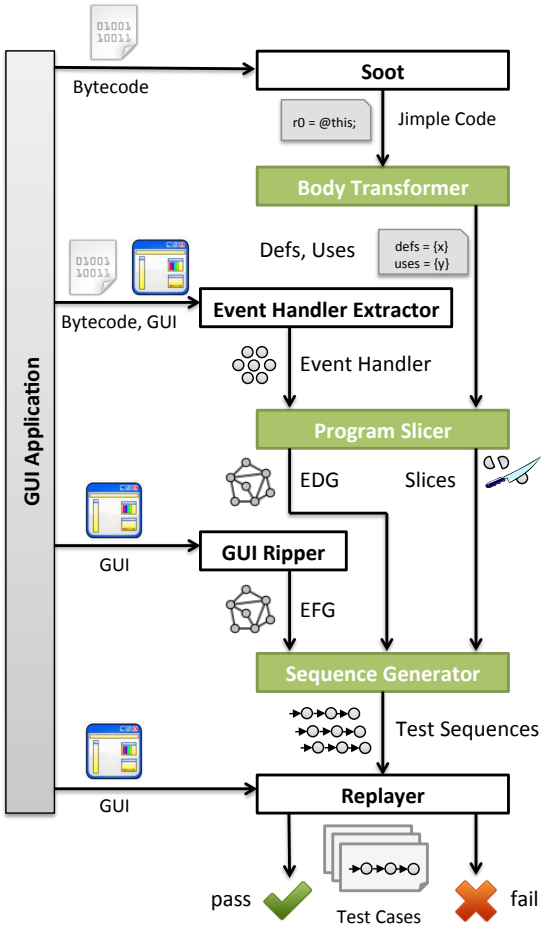


Figure 4: Our implementation is based on three main components: (1) The *Body Transformer* collects all defs and all uses from each (method) body. (2) The *Program Slicer* constructs the EDG data structure and creates a def-use chain (i. e., a program slice) for each field definition (field-def). (3) The *Sequence Generator* generates a set of event sequences from the EDG and uses the slices to eliminate redundant event sequences.

As an example, Figure 5 shows the Jimple code for the event handler `onSlider` presented in the motivating example (see Figure 1). Here, the body transformer collects the following defs (in the following depicted in red boxes): `r0` (line 6), `$r1` (line 7), `i0` (line 8), `r0.angle` (line 9), `$i1` (line 10), and `$i2` (line 11). Furthermore, the body transformer collects the following uses (depicted in green boxes): `@this` (line 6), `r0.slider` (line 7), `$r1` (line 8), `i0` (line 9), `r0.convert` (line 10), `r0.angle` (line 11), `$i1` and `$i2` (both line 12).

4.3 Program Slicer

The program slicer takes as input the event handlers (that come from an additional component that extracts them accordingly), and the set of defs and uses from the body transformer. Then, the slicer both constructs the EDG (according to the technique presented in Arlt et al. [3]) and computes the set of field slices of the GUI application. For the computation of the slices, we have to take into account that an edge in the EDG reflects a def-use dependency between two

```
1  void onSlider() {
2      ModifyImageWindow r0;
3      int i0, $i1, $i2;
4      Slider $r1;
5
6      r0 := @this;
7      $r1 = r0.slider;
8      i0 = $r1.getValue();
9      r0.angle = i0;
10     $i1 = r0.convert;
11     $i2 = r0.angle;
12     r0.print($i1, $i2);
13 }
```

Figure 5: Jimple code of the event handler `onSlider` of the example GUI. The code snippet consists of 10 units (i. e., line 2-4 and line 6-12) used to extract defs and uses, and to compute backward slices.

events, rather than between two *fields* in the events. Unlike a (local) variable, the scope of a field is not restricted to only one method. In order to provide the dependency information of fields, we apply backward slicing, i. e., we compute the backward slice for each field definition (field-def).

For example, for the Jimple code depicted in Figure 5, the slicer computes the backward slice for the field-def `r0.angle` in line 9. By looking up the uses of this field-def in $Uses(u)$, the slicer detects that this field-def depends on the use of the variable `i0`. Hence, `i0` is added to the backward slice. Afterwards, the slicer continues to look up a possible definition of the variable `i0` by iterating over all units in the set $Defs(u)$. It finds the definition in line 8 and detects, again by looking up the set $Uses(u)$, that this definition depends on the use of the variable `$r1`. Hence, `$r1` is added to the backward slice as well. The slicer continues to look for further dependencies and only stops if a certain definition is not found, or the scope of the analysis is reached (i. e., the classpath of the application). In our example, the backward slice for the field-def `r0.angle` consists of `i0`, `$r1`, and `r0.slider`.

4.3.1 Interprocedural Slicing

Our implementation supports interprocedural slicing. In particular, we recursively follow all methods called in an event handler method. For example, if a variable is defined by the return value of a called method, then the slicer follows the method and performs a backward slice of the return statement of the called method. However, following method calls is difficult in particular when class inheritance is used. For example, if a method call is polymorphic (that is, *virtual* in Java), the slicer has to identify the actual object that calls the method. In order to overcome this technical hurdle, we apply SPARK [29], the interprocedural points-to-analysis of Soot. SPARK returns a set of possible objects that may have called a particular method. In our implementation, we first analyze only those methods that belong to the possible objects retrieved by SPARK. Afterwards, we aggregate the slices of each analyzed method to the slice of the corresponding event handler method.

4.4 Sequence Generator

In the previous section we have discussed how the body transformer and the program slicer compute slices of events. In this section we will discuss how the slices are applied by the sequence generator component in order to eliminate redundant event sequences.

The sequence generator takes as input the EDG and the slices from the program slicer (as well the Event Flow Graph from the GUI Ripper, see the section *Auxiliary Steps* below). It returns the set of test sequences that can be executed by the replayer component. In a first step, the sequence generator uses the EDG in order to generate all sequences of a specific length n as defined in Section 3. In a second step, the sequence generator uses the slices and performs the computation and elimination of redundant event sequences. In a future version of the implementation, we will investigate the feasibility of combining these two steps into one single step.

4.4.1 Computation of PO-redundant Sequences

In order to detect whether the ordering of two events e and e' does not matter to test an event e'' in a sequence, the sequence generator analyzes if the intersection of all backward slices of field-defs from e and from e' is empty. If this is the case, then e and e' are independent, and thus, the sequence $\langle e, e', e'' \rangle$ is equivalent to the sequence $\langle e', e, e'' \rangle$.

For example, in order to test the event `onOK` of the example GUI, the sequence generator may first select the two sequences $s_1 = \langle$ `onCheckBox`, `onSlider`, `onOK` $\rangle$ and $s_2 = \langle$ `onSlider`, `onCheckBox`, `onOK` $\rangle$. These two event sequences are generated because the event `onCheckBox` writes the field `convert` and the event `onSlider` writes the field `angle`, which are both read in the event `onOK`. Afterwards, the backward slice of the field `convert` in the event `onCheckBox` and the backward slice of the field `angle` in the event `onSlider` are analyzed. In this example, we get the following result (for brevity, we use the Java code instead of the Jimple code for presentation).

`convert` = {`cbValue`, `checkBox`}
`angle` = {`sliderValue`, `slider`}

We observe that the intersection of `convert` and `angle` is empty, which accurately reflects the independence of the corresponding fields `convert` and `angle`. Based on this information, we eliminate one of these sequences. In particular, we observe once again that although the EDG expresses a def-use dependency between `onCheckBox` and `onSlider`, there exists no causal data-flow between these events.

4.4.2 Computation of CV-redundant Sequences

In order to detect whether there exists no causal data-flow between three events in a sequence, say $\langle e, e', e'' \rangle$, the sequence generator analyzes if all backward slices of fields in e' do not contain any field-def of e. If this is the case, then e and e' are not dependent, and thus, the effect of e does not affect e''. Hence, the sequence is considered as redundant, because it is sufficient to test the sequences $\langle e, e' \rangle$ and $\langle e', e'' \rangle$.

For example, in order to test the event `onSave` of the example GUI, the sequence generator selects the sequence $s_3 = \langle$ `onCheckBox`, `onSlider`, `onSave` $\rangle$. This event sequence is generated because the event `onCheckBox` writes the field `convert` which is read in `onSlider`, and the event `onSlider` writes the field `angle` which is read by `onSave`. Analyzing the backward slice of the field `angle` in `onSlider` provides the following information.

`angle` = {`sliderValue`, `slider`}

We observe that the field `angle` written in `onSlider` does not depend on the field `convert` written in `onCheckBox`. Hence, the sequence is CV-redundant and thus eliminated. In particular, we observe again that it suffices to test the sequences ⟨`onCheckBox`, `onSave`⟩ and ⟨`onSlider`, `onSave`⟩ instead.

Auxiliary Steps

The event handler extractor takes as input both the bytecode and the GUI of the application. It executes the GUI application and enumerates all found widgets. For each found widget, reflection [39] is applied to obtain the corresponding Java object and its assigned event handlers.

The GUI ripper takes as input the GUI of the application and constructs an Event Flow Graph [35]. The replayer takes as input a set of test sequences, embeds each sequence into a test case, executes the test case, and finally reports its results [37].

We have implemented our technique into the tool Gazoo. The source code is publicly available [13]. Furthermore, our implementation uses an adaptation of the GUI ripper and the replayer of GUITAR [37].

5. EXPERIMENTS

In this section we experimentally evaluate our approach. We first present the setup of the experiments. Afterwards, we address a set of research questions and discuss them with the help of the results of the experiments.

5.1 Setup of the Experiments

We evaluate our approach using stable versions of six Java-based open source applications: *JabRef 2.7* [24] manages bibliographic references, *FreeMind 0.9* [11] generates mind maps, and *Rachota 2.3* [38] is a time recording system. Furthermore, the applications *TerpWord*, *TerpSpreadSheet* and *TerpPaint* form a suite of office applications developed by undergraduate students. For Rachota, TerpWord, TerpSpreadSheet, and TerpPaint, we use the artifacts available from *Community Event-based Testing (COMET)* [9]. In particular, we choose these applications to consider both small (TerpWord: 6,842 LOC) and large applications (JabRef: 77,745 LOC). Figure 6 shows some relevant statistics of each Application Under Test (AUT).

AUT	Events	Methods	Units	Defs	Uses
JabRef	776	7,930	123,760	5,980	18,206
FreeMind	959	8,111	75,547	3,392	8,459
Rachota	154	1,838	24,635	1,399	3,189
TerpPaint	317	1,759	34,323	2,524	8,474
TerpSpread.	312	1,295	15,206	722	2,013
TerpWord	159	965	10,821	368	1,538

Figure 6: Statistical information about the applications under test (AUTs) used in the evaluation of our approach.

In our experiments, we compare the performance of our slicing-based approach for $n = 3$ (in the following called *EDG-3-Sliced*) to the baseline EDG-approach without slicing (i. e., to the algorithm *GenerateRelevantTestCases* as described in Section 3 without slicing, called *EDG-3* in the following). Moreover, we compare to the approach proposed by Memon [34] (called *EFG-3* in the following)[1].

In all three configurations, we set the parameter to $n = 3$. This choice is motivated by the fact that the number of event sequences generated by a black-box approach already becomes unmanageable for sequences of length greater than 3 [51]. For example, generating all sequences of length 4 for the application JabRef would already result in more than 175 million sequences to be executed, which is clearly too large to be handled with reasonable resource limits. Still, by setting the parameter to $n = 3$, the total number of executed test cases in our experiments amounts to 49,378,177.

Each event sequence is embedded into one test case as described in Section 4. The replayer generates random input data, e. g., random strings for text boxes. The computation of suitable input data, which can possibly achieve higher code coverage (see [2, 12, 18]) represents an orthogonal problem and is not in the scope of this paper. The replayer employs a crash monitor as an oracle, which is simple but reasonable. In particular, we record any exception occurred during test case execution. For a discussion of alternative oracles, we refer to Memon et al. [36].

The test cases are executed on a cluster with 50 virtual machines, which represents a rather common experimental setup both in the research community [33] as well as in industry.[2] Each virtual machine utilizes one physical CPU-core with 2.0 GHz, 1 GB RAM, and 20 GB HDD. In order to mitigate the effect of randomness, all configurations are executed three times.

5.2 Results of the Experiments

We evaluate our approach with respect to the following research questions.

Is our approach able to eliminate a non-trivial number of redundant event sequences?

To answer this question, we refer to Figure 7, which provides an overview of our results. Apparently, for all AUTs, the configuration *EDG-3-Sliced* is able to eliminate a relatively high number of redundant sequences. In particular, for JabRef, the configuration *EDG-3-Sliced* eliminates 69% of the sequences generated by the configuration *EDG-3*. Similarly, *EDG-3-Sliced* eliminates 74% for FreeMind, 72% for Rachota and TerpPaint, 46% for TerpSpreadSheet and 41% for TerpWord. Overall, in all of these GUI applications, the number of sequences is reduced to less than a half, in 4 out of 6 applications even to roughly a quarter of the original sequences with *EDG-3*.

Does our approach scale to realistic GUI applications, particularly w.r.t. the overhead for computing the slices?

To answer this question, we again refer to Figure 7, which particularly shows the number of generated event sequences, the overall *generation time* (i. e., the time needed for generating the whole set of event sequences), and the *execution time* (i. e., the overall time needed to execute ("replay") the generated event sequences, measured on a cluster with 50 virtual machines).

First, comparing *EDG-3-Sliced* to *EFG-3*, we observe that the number of generated sequences for *EDG-3-Sliced* is significantly lower for JabRef, FreeMind and TerpPaint. For Rachota, TerpSpreadSheet and TerpWord, the configuration *EDG-3-Sliced* generates more sequences than *EFG-3*, because

[1] Internally, we use an optimized version of the EFG [51] called Event Interaction Graph (EIG).

[2] Personal communication (mid-sized software company and industrial partner)

AUT	EFG-3	EDG-3	EDG-3 Sliced
JabRef			
# sequences	19,859,369	821,993	**255,132**
generation time (m)	2,648	151	164
gen. time per seq. (ms)	8	11	12
execution time (d)	133.30	5.50	**1.70**
line coverage (%)	56	58	58
# detected bugs	🐞	🐞🐞🐞🐞	🐞🐞🐞🐞
FreeMind			
# sequences	17,830,612	6,093,201	**1,600,638**
generation time (m)	2,377	1,016	1,219
gen. time per seq. (ms)	8	10	12
execution time (d)	123.82	42.30	**11.10**
line coverage (%)	54	54	54
# detected bugs	–	–	–
Rachota			
# sequences	22,588	123,256	**34,981**
generation time (m)	3	16	18
gen. time per seq. (ms)	8	8	9
execution time (d)	0.08	0.44	**0.12**
line coverage (%)	62	64	64
# detected bugs	–	🐞	🐞
TerpPaint			
# sequences	1,098,639	495,467	**138,603**
generation time (m)	146	83	91
gen. time per seq. (ms)	8	10	11
execution time (d)	4.06	1.82	**0.50**
line coverage (%)	49	54	54
# detected bugs	–	🐞🐞	🐞🐞
TerpSpreadSheet			
# sequences	40,299	117,938	**63,184**
generation time (m)	5	16	18
gen. time per seq. (ms)	8	8	9
execution time (d)	0.12	0.34	**0.18**
line coverage (%)	45	48	48
# detected bugs	🐞	🐞	🐞
TerpWord			
# sequences	122,991	415,888	**243,398**
generation time (m)	16	62	69
gen. time per seq. (ms)	8	9	10
execution time (d)	0.36	1.24	**0.72**
line coverage (%)	56	56	56
# detected bugs	–	–	–

Figure 7: Results overview of the experiments. Abbreviations: *#sequences*: number of generated event sequences. *Generation time*: overall time (in minutes) needed to generate all sequences (including the time needed for our slicing technique in *EDG-3-Sliced*). *Execution time*: overall time (in days) needed to execute ("replay") all sequences (on a cluster with 50 virtual machines).

these applications apparently contain more depending events than consecutive (executable) events for sequences of length $n = 3$. However, the execution time of *EDG-3-Sliced* compared to *EFG-3* is still acceptable, and significantly lower than *EDG-3* without slicing in all these applications. In particular, compared to *EDG-3*, the reduced number of sequences with *EDG-3-Sliced* leads to a significant reduction of execution time. For example, the execution time for the large AUTs JabRef and FreeMind is reduced by several days (3.8 days for JabRef and 31.2 days for FreeMind). Considering the overall resulting execution time, FreeMind (where 11.1 days are needed to execute the 1,600,638 test cases) is an outlier. Furthermore, with *EDG-3-Sliced*, we obtain the same code coverage as with *EDG-3*. We will discuss both *execution time* and *code coverage* in more detail in Section 5.3.

Second, considering the overhead to compute program slices, let us have a closer look at the generation time. We observe that the generation time *per sequence* of *EDG-3-Sliced* compared to *EDG-3* and *EFG-3* incurred by program slicing is not an issue in practice: For all AUTs, the occurred overhead consists of 2–4 milliseconds. This is probably best explained by Figure 8, which depicts the lengths of the backward slices occurred in the analysis: While there are some outliers (up to length 304 for TerpPaint), the mean length of the slices is short (i. e., in a range of 5 to 7) and can be computed efficiently.

AUT	Min	Median	Mean	Max
JabRef	1	3	7	191
FreeMind	1	3	6	217
Rachota	2	4	6	206
TerpPaint	1	3	7	304
TerpSpreadSheet	2	3	5	58
TerpWord	2	3	6	31

Figure 8: Statistical information about the lengths (number of statements) of the backward slices computed for the AUTs in our experiments.

What is the proportion of PO-redundant and CV-redundant sequences compared to the overall number of redundant sequences?

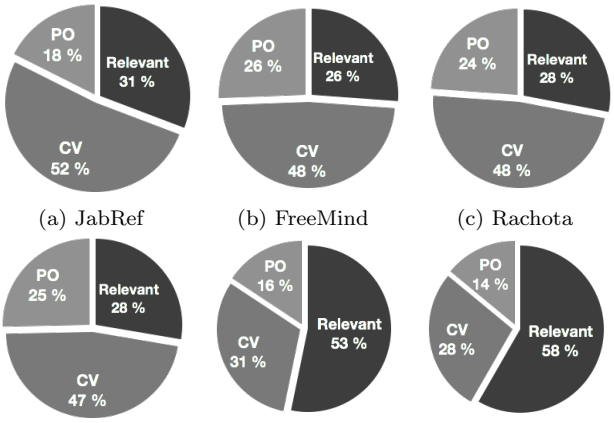
To answer this question, we refer to Figure 9, which shows the corresponding proportions graphically for all AUTs. In this figure, sequences that are neither PO-redundant nor CV-redundant are called *relevant*. We observe that CV-redundant sequences occur in 36%-55% of the cases, whereas PO-redundant sequences range from 18%-25%. That is, causal event sequences of length 3 do not appear frequently in the GUI applications of our experiments. A potential explanation is that most of the event handlers are coherent and not strongly coupled to other events. For example, it is likely that the events in a search-and-replace window do not affect the events in the preference window, and vice versa. Overall, in the considered applications under test, CV-redundant sequences occur slightly more often than PO-redundant ones.

How do the lengths of generated event sequences with *EDG-3-Sliced* compare to those with *EDG-2*?

To answer this question in more detail, we refer to Figure 10, which shows the length of the sequences (x-axis) and the number of generated event sequences (y-axis, in log scale). We present detailed data for the small application TerpWord and for the two large applications JabRef and FreeMind (the results for the other three applications are similar). The black points indicate the number of sequences generated by *EDG-3-Sliced*; the gray points indicate the number of sequences generated by *EDG-2* [3]. We observe that for most n, *EDG-3-Sliced* produces significantly more sequences of length n. Furthermore, we observe that *EDG-3-Sliced* handles significantly longer sequences than *EDG-2*. For example, in FreeMind, the length of the longest generated sequences increases from 10 (*EDG-2*) to 17 (*EDG-3-Sliced*). Note that, in order to capture all these sequences with a black-box approach, the parameter n for *EFG-n* would have to be increased even more, causing a much larger number of overall test cases.

Is our approach effective in detecting bugs?

To answer this question, we again refer to Figure 7. Overall, we detected four bugs in JabRef with the configuration *EDG-3* and *EDG-3-Sliced*, respectively, compared to one detected bug with *EFG-3*. In Rachota, we detected one bug only with *EDG-3* and *EDG-3-Sliced*. Furthermore, we detected two



(a) JabRef (b) FreeMind (c) Rachota

(d) TerpPaint (e) TerpSpreadSheet (f) TerpWord

Figure 9: The proportions of *redundant* sequences eliminated by the techniques PO redundancy and CV redundancy, as well as the proportion of remaining *relevant* sequences.

bugs in TerpPaint. In TerpSpreadSheet we detected one bug with *EFG-3*, *EDG-3* and *EDG-3-Sliced*. Note that these results testify our theoretical results from the last section: We detect the same set of bugs given we use the same oracle in all configurations. We remark that all detected bugs are confirmed bugs and are reported to the corresponding developers (and have been fixed in the meanwhile).

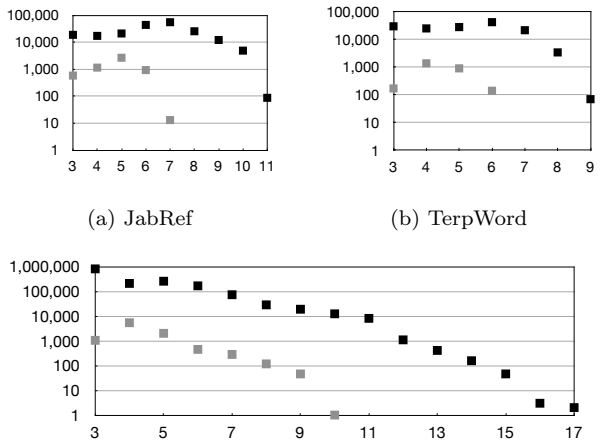
5.3 Discussion

We discuss the results considering the execution time and the code coverage in more detail.

5.3.1 Execution Time

The execution time of test cases generally is a resource critical factor in software testing. In particular, the execution time is critical for GUI testing, because replaying the test cases is particularly expensive in this context. For example, the replayer must pause roughly 500 milliseconds after an event is triggered in order to wait until the GUI toolkit has "painted" the (possibly new) set of widgets. Hence, the detection and elimination of unnecessary event sequences becomes particularly important in this setting.

From the experiments, we have observed that the number of event sequences with *EDG-3-Sliced* could be significantly reduced, and the reduced number of test cases also translated to a reduction of overall execution time. In particular, we have seen that *EDG-3-Sliced* needs less time than *EDG-3* for all AUTs, and *EDG-3* often occupies less time than *EFG-3* for JabRef, FreeMind, and TerpPaint. For Rachota, TerpSpreadSheet, and TerpWord, *EDG-3* occupies more time than *EFG-3*, but is able to generate longer sequences at the same time (see Figure 10). Apparently, FreeMind is an outlier, because it occupies more than 10 days of overall execution time even for *EDG-3-Sliced*. However, FreeMind is a rather large real-world application, and our reduction techniques considerably reduce the number of 6,093,201 sequences obtained with *EDG-3* – the execution time with *EDG-3-Sliced* for the resulting 1,600,638 test cases is still considerably lower (i.e., reduced by 31.2 days) compared to *EDG-3*.



(a) JabRef (b) TerpWord

(c) FreeMind

Figure 10: The distribution of sequence lengths. The x-axis stands for the length of the sequences. The y-axis (in log scale) represents the number of generated sequences. The black points indicate the sequences generated by *EDG-3-Sliced*; the gray points indicate the sequences generated by *EDG-2*.

5.3.2 Code Coverage

In Figure 7 we particularly report the line coverage, which apparently is relatively low for all AUTs. The main reason for this is the selection of input data: In our experiments, we use a random input selection strategy (e. g., by randomly selecting strings for text boxes) in order to avoid the computational overhead induced by more informed strategies like symbolic execution [15]. Although the selection of more suitable input data improves the code coverage [2, 12, 15, 18] (since GUI applications typically evaluate widgets that accept input data), we argue that this is an orthogonal research problem to the problem addressed in this work. Generally, our approach can be combined with arbitrary (and particularly, with more informed) strategies for selecting the input data, and still can retain the resulting (potentially higher) code coverage.

5.4 Threats to Validity

The first threat to internal validity is the creation of the model of the GUI application, namely the Event Flow Graph (EFG). Since the EFG is created using a GUI ripper, it is not guaranteed that all events of the application are found. Hence, the used EFG represents an approximation of the actual event-flow of the GUI application. That is, a path in the EFG (an event sequence) might not be executable, since its events are pair-wise executable. However, there is empirical evidence that even long event sequences can run without failures [50]. We remark that the evaluation of non-executable event sequences is not in the scope of this paper, and refer to Bae et al. [6] for a discussion on the strengths of different approaches to the creation of GUI models.

A further threat to internal validity is the replication of the experiments, because the applications under test depend on influences from "outside": For example, if the GUI application stores user settings (e. g., recently opened files) after the execution of a test case, then these user settings have to be

deleted before running the next test case. Otherwise the test case may mistakenly fail (e. g., the GUI differs from the previous test case). Moreover, some applications strongly depend on the date and time in the moment of their execution (e. g., calendar widgets). In order to decrease these threats to internal validity, each virtual machine is assigned to the same date and time, and is reset to a common state before running a new test case.

A threat to external validity is the approximation used in our implementation, which does currently not support all features of Java. For example, we currently do not support reflection [39], which allows the modification of the program's behavior at run-time. As a consequence, our implementation might wrongly keep redundant sequences. Furthermore, in our implementation, we perform *static* slicing [45] (rather than *dynamic* slicing [1, 27]). The choice is motivated by the efficiency/precision tradeoff between static and dynamic slicing. That is, while static slicing does not require to execute a corresponding program, it loses precision comparing to dynamic slicing. For example, when computing backward slices, our implementation ignores the dedicated `@this` statement in the Jimple code, whereas this statement would be mapped to the actual (owner) object of the method when performing dynamic slicing. Hence, although our theoretical framework guarantees to not wrongly exclude non-redundant sequences, our approximative implementation does not provide such a guarantee. However, as our experimental results have shown, this appears to be not an issue in practice, where we have been able to effectively reduce large test suites on the one hand, and did neither lose coverage nor test effectiveness on the other hand.

6. RELATED WORK

GUI testing is an active research area, and various test case generation approaches have been proposed [3, 8, 10, 12, 18, 31, 34, 42, 46]. These approaches can be classified in many different ways, including iterative [18, 31] and non-iterative [3, 8, 12, 34, 42, 46] as well as white-box [12, 42] and black-box approaches [8, 18, 31, 34, 46]. The proposed approach of this paper is orthogonal to both iterative and non-iterative test case generation: Although used in a non-iterative setting in this work, our techniques to reduce the number of test cases can be applied in iterative settings as well; we leave a more detailed investigation for future work. Furthermore, our overall approach both contains white-box and black-box components and is an effective generalization of a recent static analysis approach [3].

As already discussed, a particular bottleneck in GUI testing is the execution time of the generated test cases. Hence, suitable techniques to minimize the resulting suite are desired. The minimization technique proposed in this paper specifically exploits the information gained from graphs that reflect the dependencies of GUI events – to the best of our knowledge, there are no further minimization approaches in this setting. In contrast, for classical testing, there has been major efforts on the minimization of existing test suites (e. g., [22, 26, 40, 41, 47, 49]).

For these minimization approaches, however, fewer test cases cause a lower bug detection rate: For example, Jeffrey and Gupta [26] explicitly keep some "redundant" test cases in a test suite in order to be more effective in fault detection. In contrast, our slicing-based implementation indicates to obtain the same code coverage and to find the same bugs (using the same oracle) with the reduced test suite, as we use informed strategies of partial order reduction and causal variable analysis to eliminate redundant event sequences.

Program slicing was first introduced by Weiser [45]. Slicing based approaches can be classified in static slicing [45] and dynamic slicing [1, 27]. Both static and dynamic slicing inherit their own pros and cons [21, 43] in terms of precision and efficiency. In our setting, static slicing is the suitable choice as our current framework is non-iterative (as discussed above) and does not make assumptions on the program's input. In general, the range of program input (e. g., characters for a text box) is prohibitively large in GUI testing. We remark that an integration of our approach into iterative test case generation algorithms will possibly require dynamic slicing as well; again, we leave a more detailed discussion for future work. Overall, program slicing is a well-established technique and has been successfully applied in various areas of computer science like program debugging, software maintenance, reverse engineering, compiler optimization, and software testing [7, 19, 20, 27]. In the context of software testing, program slicing has mainly been used for regression testing, e. g., to detect obsolete test cases in an existing test suite. To the best of our knowledge, slicing-based approaches have not been investigated for GUI testing so far.

We remark that tool support for program slicing is available. For example, Indus [25] is a slicer for concurrent Java programs. However, Indus is limited to Java 1.4-compatible bytecode. Since today's applications (e. g., the AUTs, the ripper, and the replayer) require newer Java versions, we have implemented a new program slicing tool in order to handle Java-7-compatible bytecode. The new tool represents a lightweight implementation of static program slicing, and is tailored to the specific requirement of event sequence generation.

7. CONCLUSION

We have presented an approach to identify and to eliminate redundant event sequences in GUI test suites via program slicing. Our experimental evaluation shows that redundant event sequences can be computed efficiently and often occur in real-world GUI applications. Considering performance, we have demonstrated that eliminating redundant event sequences can greatly speed-up the overall execution time of the resulting GUI test suite.

Clearly, GUI test suite reduction is a challenge not only for non-iterative GUI testing. Hence, for the future, a promising direction for research is to investigate our techniques for iterative approaches, e.g., EXSYST [18] and AutoBlackTest [31]. A further direction will be to investigate whether our techniques can help in the context of regression testing, e. g., when a set of existing test cases needs to be maintained [10, 17, 32].

8. ACKNOWLEDGMENTS

We kindly thank the Horst Klaes GmbH & Co. KG for providing resources that allowed us to evaluate our approach. This work is supported by the Fonds National de la Recherche, Luxembourg (FNR/P10/03 - Verification and Validation Laboratory), and by the Swiss National Science Foundation (SNSF) as part of the project "Safe Pruning in Optimal State-Space Search (SPOSSS)".

9. REFERENCES

[1] H. Agrawal and J. R. Horgan. Dynamic Program Slicing. In *PLDI*, pages 246–256, 1990.

[2] S. Arlt, P. Borromeo, M. Schäf, and A. Podelski. Parameterized GUI Tests. In *ICTSS*, pages 247–262, 2012.

[3] S. Arlt, A. Podelski, C. Bertolini, M. Schäf, I. Banerjee, and A. M. Memon. Lightweight Static Analysis for GUI Testing. In *ISSRE*, pages 301–310, 2012.

[4] ASM. A Java bytecode manipulation and analysis framework. `http://asm.ow2.org`, Sept. 2013.

[5] R. A. Assi and W. Masri. Identifying Failure-Correlated Dependence Chains. In *ICST Workshops*, pages 607–616, 2011.

[6] G. Bae, G. Rothermel, and D.-H. Bae. On the relative strengths of model-based and dynamic event extraction-based gui testing techniques: An empirical study. In *ISSRE*, pages 181–190, 2012.

[7] S. Bates and S. Horwitz. Incremental Program Testing Using Program Dependence Graphs. In *POPL*, pages 384–396, 1993.

[8] F. Belli. Finite-State Testing and Analysis of Graphical User Interfaces. In *ISSRE*, pages 34–43, 2001.

[9] COMET. Community Event-based Testing. `http://comet.unl.edu`, Sept. 2013.

[10] P. Devaki, S. Thummalapenta, N. Singhania, and S. Sinha. Efficient and flexible gui test execution via test merging. In *ISSTA*, pages 34–44, 2013.

[11] FreeMind. A mind mapping software. `http://freemind.sourceforge.net`, Sept. 2013.

[12] S. R. Ganov, C. Killmar, S. Khurshid, and D. E. Perry. Event Listener Analysis and Symbolic Execution for Testing GUI Applications. In *ICFEM*, pages 69–87, 2009.

[13] Gazoo. A tool that generates relevant event sequences for GUI test cases. `http://gazoo.informatik.uni-freiburg.de`, Sept. 2013.

[14] P. Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems - An Approach to the State-Explosion Problem*, volume 1032 of *Lecture Notes in Computer Science*. Springer, 1996.

[15] P. Godefroid, N. Klarlund, and K. Sen. DART: directed automated random testing. In *PLDI*, pages 213–223, 2005.

[16] P. Godefroid, D. Peled, and M. G. Staskauskas. Using Partial-Order Methods in the Formal Validation of Industrial Concurrent Programs. In *ISSTA*, pages 261–269, 1996.

[17] M. Grechanik, Q. Xie, and C. Fu. Maintaining and evolving GUI-directed test scripts. In *ICSE*, pages 408–418, 2009.

[18] F. Gross, G. Fraser, and A. Zeller. Search-based system testing: high coverage, no false alarms. In *ISSTA*, pages 67–77, 2012.

[19] R. Gupta, M. J. Harrold, and M. L. Soffa. Program Slicing-Based Regression Testing Techniques. *Softw. Test., Verif. Reliab.*, 6(2):83–111, 1996.

[20] M. Harman and S. Danicic. Using Program Slicing to Simplify Testing. *Softw. Test., Verif. Reliab.*, 5(3):143–162, 1995.

[21] M. Harman and R. M. Hierons. An overview of program slicing. *Software Focus*, 2(3):85–92, 2001.

[22] M. J. Harrold, R. Gupta, and M. L. Soffa. A Methodology for Controlling the Size of a Test Suite. *ACM Trans. Softw. Eng. Methodol.*, 2(3):270–285, 1993.

[23] P. Haslum and H. Geffner. Admissible Heuristics for Optimal Planning. In *AIPS*, pages 140–149, 2000.

[24] JabRef. A bibliography reference manager. `http://jabref.sourceforge.net`, Sept. 2013.

[25] G. Jayaraman, V. P. Ranganath, and J. Hatcliff. Kaveri: Delivering the Indus Java Program Slicer to Eclipse. In *FASE*, pages 269–272, 2005.

[26] D. Jeffrey and N. Gupta. Test Suite Reduction with Selective Redundancy. In *ICSM*, pages 549–558, 2005.

[27] M. Kamkar, P. Fritzson, and N. Shahmehri. Interprocedural Dynamic Slicing Applied to Interprocedural Data Flow Testing. In *ICSM*, pages 386–395, 1993.

[28] P. Lam, E. Bodden, O. Lhoták, and L. Hendren. The Soot framework for Java program analysis: a retrospective. In *Cetus Users and Compiler Infrastructure Workshop*, Galveston Island, TX, October 2011.

[29] O. Lhoták and L. J. Hendren. Scaling java points-to analysis using spark. In *CC*, pages 153–169, 2003.

[30] T. Lindholm and F. Yellin. *Java Virtual Machine Specification*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd edition, 1999.

[31] L. Mariani, M. Pezzè, O. Riganelli, and M. Santoro. AutoBlackTest: Automatic Black-Box Testing of Interactive Applications. In *ICST*, pages 81–90, 2012.

[32] S. McMaster and A. M. Memon. Call stack coverage for gui test-suite reduction. In *ISSRE*, pages 33–44, 2006.

[33] A. Memon. Large scale test suites running live with GUITAR. `http://samwise.cs.umd.edu:8080/`, Sept. 2013.

[34] A. M. Memon. An event-flow model of GUI-based applications for testing. *Softw. Test., Verif. Reliab.*, 17(3):137–157, 2007.

[35] A. M. Memon, I. Banerjee, and A. Nagarajan. GUI Ripping: Reverse Engineering of Graphical User Interfaces for Testing. In *WCRE*, pages 260–269, 2003.

[36] A. M. Memon, I. Banerjee, and A. Nagarajan. What Test Oracle Should I Use for Effective GUI Testing? In *ASE*, pages 164–173, 2003.

[37] A. M. Memon and M. B. Cohen. Automated testing of gui applications: models, tools, and controlling flakiness. In *ICSE*, pages 1479–1480, 2013.

[38] Rachota. An application for timetracking projects. `http://rachota.sourceforge.net`, Sept. 2013.

[39] Reflection. The Reflection API. `http://docs.oracle.com/javase/tutorial/reflect/`, Sept. 2013.

[40] G. Rothermel, M. J. Harrold, J. Ostrin, and C. Hong. An Empirical Study of the Effects of Minimization on the Fault Detection Capabilities of Test Suites. In *ICSM*, pages 34–43, 1998.

[41] G. Rothermel, R. H. Untch, C. Chu, and M. J. Harrold. Prioritizing Test Cases For Regression Testing. *IEEE Trans. Software Eng.*, 27(10):929–948, 2001.

[42] J. C. Silva, C. E. Silva, R. D. Gonçalo, J. Saraiva, and

J. C. Campos. The GUISurfer tool: towards a language independent approach to reverse engineering GUI code. In *EICS*, pages 181–186, 2010.

[43] F. Tip. A survey of program slicing techniques. *J. Prog. Lang.*, 3(3), 1995.

[44] R. Vallée-Rai, P. Co, E. Gagnon, L. J. Hendren, P. Lam, and V. Sundaresan. Soot - a Java bytecode optimization framework. In *CASCON*, page 13, 1999.

[45] M. Weiser. Program Slicing. In *ICSE*, pages 439–449, 1981.

[46] L. J. White and H. Almezen. Generating Test Cases for GUI Responsibilities Using Complete Interaction Sequences. In *ISSRE*, pages 110–123, 2000.

[47] W. E. Wong, J. R. Horgan, S. London, and A. P. Mathur. Effect of Test Set Minimization on Fault Detection Effectiveness. In *ICSE*, pages 41–50, 1995.

[48] Q. Xie and A. M. Memon. Studying the characteristics of a "good" gui test suite. In *ISSRE*, pages 159–168, 2006.

[49] T. Xie, D. Marinov, and D. Notkin. Rostra: A Framework for Detecting Redundant Object-Oriented Unit Tests. In *ASE*, pages 196–205, 2004.

[50] X. Yuan, M. B. Cohen, and A. M. Memon. Covering array sampling of input event sequences for automated gui testing. In *ASE*, pages 405–408, 2007.

[51] X. Yuan and A. M. Memon. Using GUI Run-Time State as Feedback to Generate Test Cases. In *ICSE*, pages 396–405, 2007.