Information Processing Letters 133 (2018) 49–55

Contents lists available at ScienceDirect

Information Processing Letters

www.elsevier.com/locate/ipl

Aiding exploratory testing with pruned GUI models

Jacinto Reis [*], Alexandre Mota [*]

Centro de Informática, Universidade Federal de Pernambuco, Av. Jornalista Aníbal Fernandes, s/n, Cidade Universitária, CEP 50.740-560, Recife, PE, Brazil

A R T I C L E   I N F O

Article history:
Received 27 July 2017
Received in revised form 21 November 2017
Accepted 23 January 2018
Available online 3 February 2018
Communicated by J.L. Fiadeiro

Keywords:
Software engineering
GUI testing
Static analysis
Exploratory testing
Change request

A B S T R A C T

Exploratory testing is a kind of software testing approach that emphasizes tester's experience to maximize the chances to find bugs within a specific time period. It is naturally a GUI-oriented testing activity for GUI-based systems. We propose aiding exploratory testing by providing a GUI model of the region impacted by the most recent internal code changes. We create such a delimited GUI model by pruning an original GUI model, quickly built by static analysis. This pruned GUI model is result of a reachability analysis between GUI elements and internal source code changes. Only related GUI elements are preserved. To illustrate the idea, we consider five GUI applications found in public repositories with varying changes among them. We present experimental data concerning two executions of two exploratory testing sessions: one without using our proposal and another with our proposal. For both testers, our proposal showed coverage gain in experimental data.

1. Introduction

Nowadays, applications based on a graphical user interface (GUI) are ubiquitous. Testing such applications are becoming harder, especially due to the huge state space (possible interactions). Exploratory testing [1] is seen as one of the most successful software testing approaches in this case because it is based on the freedom of experienced testers that try to exercise potentially problematic regions of a system very quickly using their expertise.

To improve an exploratory testing session, exploratory testing usually focuses on unstable test scenarios by manually examining change requests (CRs) related to most recent bug fixes and/or software improvements. However, in most cases, the information gathered from such reports may not be accurate enough to determine which intermediate GUI elements (for example, windows, buttons, and text fields) may be exercised to reach the affected GUI elements. This creates a gap between GUI elements and internally changed elements.

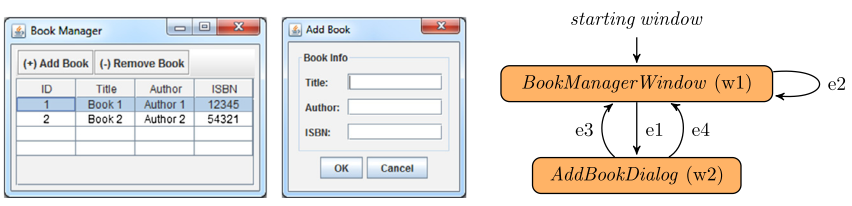
To reduce such a gap we employ Model-based GUI Testing, which is a promising research field [2–4]. We provide a pruned GUI model (related to the changed regions) as an additional source of information to aid exploratory testing sessions. This pruned GUI model emerges by keeping only those GUI elements related to internally changed elements. In other words, a full GUI model of a system, created by static analysis, is filtered by using a transitive closure operation (or reachability analysis). As illustrated in Section 6, our experiments show that our proposed strategy brings promising results, increasing the covered region in all evaluations.

The main contributions of this paper are: (i) How to create a GUI model from Java/Swing source code using Soot,[1] (ii) Soot patterns related to Swing, and (iii) Pruning GUI model based on changed code.

[*] Corresponding author.
E-mail addresses: jfsr@cin.ufpe.br (J. Reis), acm@cin.ufpe.br (A. Mota).

[1] A framework for analyzing and transforming Java and Android Applications – https://sable.github.io/soot/.

https://doi.org/10.1016/j.ipl.2018.01.008
0020-0190/© 2018 Elsevier B.V. All rights reserved.



ID	Title	Author	ISBN
1	Book 1	Author 1	12345
2	Book 2	Author 2	54321

(a) A `Book Manager` application (b) GUI Model of `Book Manager`

Fig. 1. `Book Manager` application and its GUI model.

```
public static void main(String[] args){
    Foo f = new Foo();
    int a = 7;
    System.out.println(f.bar(a));
}
```

(a) Java snippet code of a `main` method

```
public static void main(java.lang.String[]) {
    java.lang.String[] args;
    Foo $r0;
    java.io.PrintStream $r1;
    int $i0;
    args := @parameter0: java.lang.String[];
    $r0 = new Foo;
    specialinvoke $r0.<Foo: void <init>()>();
    $r1 = <java.lang.System: java.io.PrintStream out>;
    $i0 = virtualinvoke $r0.<Foo: int bar(int)>(7);
    virtualinvoke $r1.<java.io.PrintStream: void println(int)>($i0);
    return;
}
```

(b) Corresponding Jimple code

Fig. 2. Comparison between Java and Jimple code.

This paper is organized as follows. Section 2 presents our proposed GUI model. We show and explain source code patterns for Java/Swing in Section 3. Section 4 explains algorithms used to build GUI model. Section 5 describes how the changed code was reflected in pruned GUI model. In Section 6, we evaluate our proposed approach, besides discussing the obtained results. Finally, related works and conclusions are considered in Sections 7 and 8, respectively.

2. Background

This section introduces basic information about our proposed GUI Model (Section 2.1) and static analysis (Section 2.2).

2.1. GUI model

In the literature, there are different ways for representing a GUI application in terms of a mathematical model. In our work, we build the GUI model in terms of small parts, such as: *Component*, *Window* and *Event*. In short, our GUI elements are described as follows:

- An *Event* (E) is just a set of actions, like "press a button";
- A *Component* is either a *Container* or a *Widget*. We abstract away from the internal details of *Container*'s and *Widget*'s.

  $$Component ::= Widget \mid Container\langle\langle \mathbb{F}\ Component\rangle\rangle$$

- A *Window* (W) is a set of components, or formally $W \subseteq \mathbb{F}\ Component$.

After showing the elements, we present the definition of the GUI Model.

Definition 1. A GUI Model G is a 4-tuple (W, E, SW, TR), such that:

1. W is a finite set of *Window* elements;
2. E is a finite set of *Event* elements;
3. SW is a set of starting windows ($SW \subseteq W$);
4. $TR \subseteq W \times E \times W$ is a transition relation.

We exemplify our proposed GUI model using the simple application illustrated in Fig. 1a. Our model is represented by a directed graph (Fig. 1b). It illustrates our GUI Model obtained from a `Book Manager` app. Each displayed *Window* is represented as a node, where some of these windows are starting windows ($BookManagerWindow$ node in this example is a starting window). Event $e1$ expresses a click on the `Add Book` button. This action is available on main window (node $w1$), and after it is triggered it opens the `Add Book` dialog, captured by node $w2$. The events $e3$ and $e4$, which are exit events of this dialog window, represent the possibilities of clicking on buttons `OK` and `Cancel`, respectively. They are connected to the main window because after they are fired, the dialog is closed and the focus returns to `Book Manager` window $(w1)$. The remaining event $(e2)$ encodes the action of removing books. As it does not open another window, both source and target windows are the same $(w1)$.

2.2. Static analysis using Soot

To build such a GUI Model, we perform a static analysis implemented using the Soot framework. This analysis deals with the Jimple [5] intermediate representation of the Java bytecode [6] of the application under analysis. Fig. 2a shows a Java snippet of a `main` method and Listing 2b shows its Jimple corresponding code.