Finding Errors in Multithreaded GUI Applications

Sai Zhang
University of Washington
Joint work with: Hao Lu, Michael D. Ernst
GUIs are everywhere in modern software
Multithreading in GUI applications

A GUI Application

UI event 1
UI event 2

Thread #1

Thread #2

UI thread
The Single-GUI-Thread Rule

All GUI objects must be **exclusively** accessed by the **UI thread**

- Required by: SWT, Eclipse, Swing, Java, Android, Cocoa, Mac OS X, Qt, X Windows, ...

A GUI Application

UI thread

This **non-UI thread** must **not** access any GUI objects
Violation of the Single-GUI-Thread rule

- Triggers an “Invalid Thread Access Error”
- May abort the whole application
An Example Violation

public void setText(String) {
    checkThread();
    ...
}

button’s event handler:

public void runTask() {
    Runnable r = new Runnable() {
        public void run() {
            ... //do some lengthy computation
            button.setText("Finished");
        }
    };
    new thread(r).start();
}

Create a new, non-UI thread

Access the button object to set text

//GUI framework code
public void setText(String) {
    checkThread();
    ...
}

Trigger an invalid-thread-access-error
Invalid Thread Access Errors in practice

• **Pervasive**
  – One of the *top 3* bug categories in SWT [Shanmugam 2010]
  – A Google search returns *11800+* entries (bug reports, FAQs, etc.)
  – In Eclipse
    • *2732* bug reports
    • *156 confirmed* bugs in 20+ projects, 40+ components

• **Severe**
  – Often aborts the whole application

• **Hard to debug**
  – Non-trivial effort to fix (e.g., *2 years* to resolve one bug in Eclipse)
Why the Single-GUI-Thread Rule?

• Simpler programming
  – No datarace nor deadlock on GUI objects

• Less overhead
  – No locking when accessing GUI objects

• A single event queue can dispatch UI events
  – Easy event processing, program comprehension, and testing
**Our Error Detection Technique**

1. Call graph construction
2. Error detection
3. Error filtering

---

**Static Analyses**

- **A GUI Application**
- **Warnings**
- **Bugs**
- **False Positives**

- **9 applications** from 4 supported GUI frameworks
- **Less than 5 mins** per application
- **20 warnings**
- **10 bugs**
- **10 false positives**
- **5 hours human inspection**

---

**Automated**: no need for a test harness

**General**: instantiated it for **4 GUI frameworks:** SWT

**Scalable**: evaluated on 9 applications, over **1.4 M LOC** with lib code

**Practical**: found **10 bugs** with **10 false positives**
Existing Solutions for this Problem

- Testing
  - Misses many corner cases in practice

- Stylized programming rules

```java
public void runTask() {
    Runnable r = new Runnable() {
        public void run() {
            // do some lengthy computation
            button.setText("Finished");
        }
    }
    new thread(r).start();
}
```

- **Unnecessary**: if already on the UI thread
- **Dangerous**: may introduce new concurrency errors

Results on 9 evaluation programs

<table>
<thead>
<tr>
<th></th>
<th>#Warnings</th>
<th>#Bugs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Requiring Wrappers</td>
<td>6393</td>
<td>?</td>
</tr>
<tr>
<td>Our technique</td>
<td>20</td>
<td>10</td>
</tr>
</tbody>
</table>
Outline

• Problem
• Error detection technique
• Implementation
• Experiments
• Related work
• Conclusion and future work
Terminology

• **UI thread**: a single special thread to handle UI events

• **Non-UI thread**: other threads

• **UI-accessing method**: a method whose execution may read or write a GUI object

• **Safe UI method**: message-passing methods to execute code on the UI thread

```java
void runTask() {
    ...
    button.setText("..");
}
```
Assumptions

• Single UI thread

• Thread spawning:
  – Every non-UI thread is (transitively) spawned by the UI thread
Problem formulation: call graph reachability

- An invalid thread access error occurs when:
  
a non-UI thread invokes a UI-accessing method without going through a Safe UI method

- A reachability problem
  - **Source**: non-UI thread spawning
  - **Sink**: UI-accessing method

Diagram:
- **Non-UI thread spawning (source)**
- **UI-accessing method (sink)**
- **Safe UI method**
- **Other method**

```java
Thread.start()
Display.asyncExec(...)  
button.setText("")
Thread.start()
runTask()
```
Error detection algorithm

1. Construct **a call graph** for the tested program
2. Find paths from `Thread.start()` to UI-accessing methods *without* going through a safe UI method
Reflection in Constructing Call Graphs

Android Application:

```xml
<LinearLayout>
    <Button android:id="@+id/button_id" android:text="A Button" />
</LinearLayout>
```

... Button button = (Button) findViewById("button_id"); button.setText("This is a button"); ...

- **findViewById** does not explicitly construct a button object
- A call graph construction algorithm may:
  - fail to conclude the variable `button` points to a `concrete` object
  - exclude a `setText` edge to the call graph (that should exist)
  - miss 1 bug in our experiments
Reflection-aware call graph construction

Android Application:

```xml
<LinearLayout>
    <Button android:id="@+id/button_id" android:text="A Button" />
</LinearLayout>
```

**Before transformation:**

```java
Button button = (Button) findViewById("button_id");
button.setText("This is a button");
```

**After transformation:**

```java
Button button = new Button(null);
button.setText("This is a button");
```

- **Program transformation**: replace reflection calls with explicit object creation expressions
- Use an **off-the-shelf call graph construction algorithm** on the **transformed** program
Annotation support for native methods

- Relationships between native methods and Java methods

```java
@CalledByNativeMethods(callers = {"init"})
public void addTypeItem(int id, String label) { ...}
```

- Manually specified
- `addTypeItem(int, String)` may be called by native method “init”
- Our technique will miss 1 bug without such annotations
Filtering the error reports

• A static analysis may report:
  – false positives
  – redundant warnings with the same error root cause

• A set of error filters to remove likely invalid reports
  – 2 sound filters
  – 3 heuristic filters
2 sound error report filters

- **Filter 1**: remove syntactically subsumed reports
  
  \[a() \rightarrow b() \rightarrow \text{thread.start}() \rightarrow d()\]
  
  \[b() \rightarrow \text{thread.start}() \rightarrow d()\]

- **Filter 2**: remove reports containing user-annotated, project-specific methods
  
  `util.runOnUIMethod(Runnable r)`
  
  - Checks whether the current thread is UI thread or not
3 heuristic error report filters

- **Filter 3**: remove reports containing specific library calls
  
  e.g., `Runtime.shutdown`

- **Filter 4**: remove longer reports with the same “entry node $\rightarrow$ Thread.start()” head nodes
  
  $a() \rightarrow b() \rightarrow$ Thread.start() $\rightarrow c() \rightarrow d() \rightarrow e()$
  
  $a() \rightarrow b() \rightarrow$ Thread.start() $\rightarrow c() \rightarrow f()$

- **Filter 5**: remove longer reports with the same “thread.start() $\rightarrow$ ui-accessing node” tail nodes
  
  $a() \rightarrow b() \rightarrow$ Thread.start() $\rightarrow c() \rightarrow d() \rightarrow e()$
  
  $f() \rightarrow$ Thread.start() $\rightarrow c() \rightarrow d() \rightarrow e()$
Outline

• Problem
• Error detection technique
• Implementation
• Experiments
• Related work
• Conclusion and future work
Instantiation for different frameworks

• Need to customize
  – *program entry points*
  – *UI-accessing methods*
  – *Safe UI methods*
# Instantiation details for 4 GUI frameworks

<table>
<thead>
<tr>
<th>Frameworks</th>
<th>Entry node</th>
<th>UI-accessing methods</th>
<th>Safe UI method</th>
</tr>
</thead>
<tbody>
<tr>
<td>SWT</td>
<td>main()</td>
<td>checkWidget / checkDevice</td>
<td>asyncExec / syncExec</td>
</tr>
</tbody>
</table>
Outline

• Problem
• Error detection technique
• Implementation
• Experiments
• Related work
• Conclusion and future work
## Subject programs

<table>
<thead>
<tr>
<th>Programs</th>
<th>Line of Code</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>SWT desktop applications</strong></td>
<td></td>
</tr>
<tr>
<td>FileBunker</td>
<td>14,237</td>
</tr>
<tr>
<td>ArecaBackup</td>
<td>23,226</td>
</tr>
<tr>
<td><strong>Eclipse plugins</strong></td>
<td></td>
</tr>
<tr>
<td>EclipseRunner</td>
<td>3,101</td>
</tr>
<tr>
<td>HundsonEclipse</td>
<td>11,077</td>
</tr>
<tr>
<td><strong>Swing applications</strong></td>
<td></td>
</tr>
<tr>
<td>S3dropbox</td>
<td>2,353</td>
</tr>
<tr>
<td>Sudoku Solver</td>
<td>3,555</td>
</tr>
<tr>
<td><strong>Android applications</strong></td>
<td></td>
</tr>
<tr>
<td>SGTPuzzler</td>
<td>2,220</td>
</tr>
<tr>
<td>Mozilla Firefox</td>
<td>8,577</td>
</tr>
<tr>
<td>MyTracks</td>
<td>20,297</td>
</tr>
<tr>
<td><strong>Total:</strong></td>
<td><strong>89,273</strong></td>
</tr>
</tbody>
</table>

**Framework size:**
1.4 MLOC
Experimental Procedural

- Run the error detection algorithm on each application
  - 3 call graph construction algorithms
  - 2 configurations for Android applications
    - with / without call graph enhancement
      (handle reflection + annotations for native methods)

- Tool performance
  - Less than 5 minutes per application

- Manual result inspection
  - Spent 5 hours in total to check the output validity
Experimental Results

• Output 20 warnings, in which 10 are bugs (5 are new)

<table>
<thead>
<tr>
<th>Call graph algorithm</th>
<th># Warnings</th>
<th>#Bugs</th>
</tr>
</thead>
<tbody>
<tr>
<td>RTA with enhancement</td>
<td>250</td>
<td>4</td>
</tr>
<tr>
<td>0-CFA with enhancement</td>
<td>136</td>
<td>6</td>
</tr>
<tr>
<td>1-CFA with enhancement</td>
<td>20</td>
<td>10</td>
</tr>
</tbody>
</table>

• More precise call graph ⇒ more bugs found
  – 1-CFA found the most

<table>
<thead>
<tr>
<th>Call graph algorithm</th>
<th># Warnings</th>
<th>#Bugs</th>
</tr>
</thead>
<tbody>
<tr>
<td>1-CFA</td>
<td>19</td>
<td>8</td>
</tr>
<tr>
<td>1-CFA with enhancement</td>
<td>20</td>
<td>10</td>
</tr>
</tbody>
</table>

• Call graph enhancement are useful (2 more bugs)
Comparing graph search strategies

- Our technique uses **BFS**, compare it with alternatives

<table>
<thead>
<tr>
<th>Strategies</th>
<th>#Warnings</th>
<th>#Bugs</th>
</tr>
</thead>
<tbody>
<tr>
<td>BFS</td>
<td>20</td>
<td>10</td>
</tr>
</tbody>
</table>

- Observations from our subject programs
  - Multi-source BFS **omits** bugs
  - DFS searches deeper, and returns **longer** paths
    (more likely to be **invalid**, due to the conservative call graph)
  - Exhaustive search is **sound** but **infeasible** in practice
Evaluating error filters

Sound Filters:
  F1: remove lexically subsumed reports
  F2: remove annotated reports

Heuristic Filters:
  F3: remove specific library calls
  F4: merge common heads
  F5: merge common tails
Experimental conclusion

• Our technique:
  – Finds real invalid-thread-access errors
  – Detects more errors as the call graph precision increases
  – Uses BFS to find more errors than other search strategies
  – Reduces likely invalid reports via 5 error filters
Outline

• Problem
• Error detection technique
• Implementation
• Experiments
• Related work
• Conclusion and future work
Related Work

• Analyzing and testing GUI applications
  Guitar [Memon ‘00], Stabilizer [Michail ‘05], Julia [Payet ‘11] …

  Focus on test generation, error predication, and code verification; but does not support finding invalid thread access errors

• Finding bugs in multithreaded programs
  Eraser [Savage ‘97], Chord [Naik ‘05], Goldilocks [Elmas ‘07], FastTrack [Flanagan ‘09] …

  Different goals (data races + deadlocks), algorithms, and abstractions

• Call graph construction algorithms
  RTA [Bacon ‘97], k-CFA [Might ‘10], TamiFlex [Bodden ‘11] …

  Does not support reflection, or need dynamic information
Outline

• Problem
• Error detection technique
• Implementation
• Experiments
• Related work
• Conclusion and future work
Future Work

• Incorporate dynamic and symbolic analysis
  – Filter out infeasible paths
  – Identify more entry points

• Automatically fix the invalid-thread-access errors
  – Counterexample-guided reasoning
  – Heuristic reasoning

• Unit testing of multithreaded GUI applications
  – Test abstraction
  – Event simulation
Contributions

• A general *technique* to find invalid thread access errors
  – Formulate error detection as a call graph reachability problem

• A tool *implementation* supporting 4 GUI frameworks
  – Available at: https://guierrordetector.googlecode.com/

• An *evaluation* on 9 subjects shows its usefulness