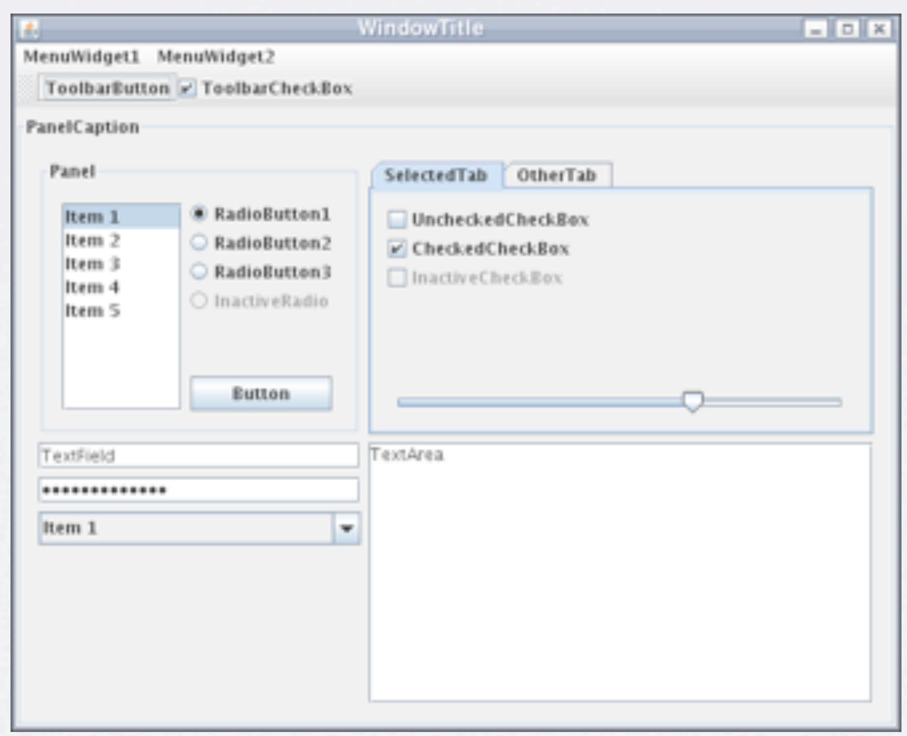
FINDING ERRORS IN MULTITHREADED GUI APPLICATIONS

Sai Zhang, Hao Lü, Micheal D. Ernst

Emanuele Rudel

MOTIVATION

Develop bug-free applications with GUI taking reflection into account



SINGLE GUI THREAD

Most G … e:

all GU … the
event …

```
Exception in thread "AWT-EventQueue-0" org.eclipse.swt.SWTException:
Invalid thread access
    at org.eclipse.swt.SWT.error(SWT.java:4083)
    at org.eclipse.swt.SWT.error(SWT.java:3998)
    at org.eclipse.swt.SWT.error(SWT.java:3969)
    at org.eclipse.swt.widgets.Display.error(Display.java:1249)
    at org.eclipse.swt.widgets.Display.checkDevice(Display.java:755)
    at org.eclipse.swt.widgets.Display.getShells(Display.java:2171)
    at org.eclipse.swt.widgets.Display.setModalDialog(Display.java:4463)
    at org.eclipse.swt.widgets.MessageBox.open(MessageBox.java:200)
```

Advantages

- no concurrency errors and overheads
- predictable behavior

FINDING INVALID THREAD ACCESS ERRORS

Hardly feasible with testing

Message passing is not always safe and predictable

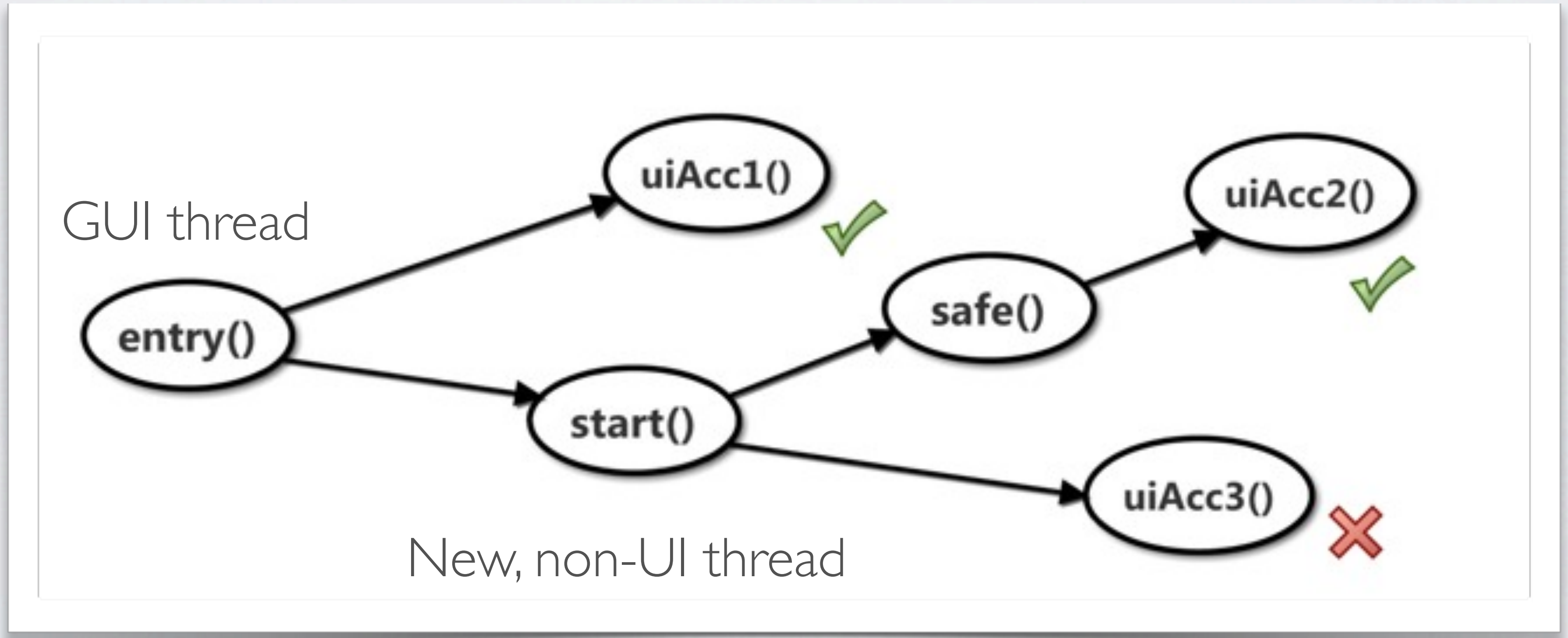
Use **static analysis**

IDEA

Given: a program and its entry nodes

Build a method call graph

Find paths accessing a UI object from a non-UI thread



ALGORITHM

```
Input: Java program P
Output: set of potential invalid thread access errors

errors := ∅
cg := construct_call_graph(P)

entry_nodes := get_entry_nodes(cg)           // starting UI thread methods
ui_accessing_nodes := ui_accessing_nodes(cg) // methods accessing UI elements
[safe_ui_nodes := ui_safe_nodes(cg)]         // methods safely accessing UI elements

foreach entry_node in entry_nodes do
  start_nodes := reachable_starts(entry_node) // starting non-UI thread methods
  foreach start in start_nodes do
    BFS(entry_node, function (a_node)
      do
         if a_node ∈ ui_accessing_nodes then
            errors ∪ error_report(a_node)
         end
      end
    )
  end for
end for

return errors
```

THE REFLECTION PROBLEM (AND SOLUTION)

Call graph algorithm (static analysis) **omits** reflection calls

Treated as null objects

```
Input: Java program P
Output: a call graph cg

foreach expression in P do
  if is_reflection_call (expression) then
    objects := may_be_created_objects (expression)
    new_expression := object_creation_expression (objects)
    replace expression with new_expression
end for
cg := construct_call_graph(P)
return cg
```

FILTERING

Filter out false positives and redundant warnings (some filters are not sound)

1. Filter lexically redundant reports

```
          a() -> b() -> c()
    d() -> a() -> b() -> c()
```

2. Filter reports with user-annotated methods

FILTERING

3. Filter reports containing library calls (e.g. shutdown)

4. Filter reports with same methods in [EntryNode, start()]

```
a() -> ... Thread.start() ... -> m() -> UIAcc1()
a() -> ... Thread.start() ... -> m() -> UIAcc2()
```

5. Filter reports with same methods in [start(), UIAcc()]
   E.g.: error in method m() which is called multiple times

99.6% of warnings removed in the experiments

EVALUATION

1. Effectiveness of the technique
2. Comparison of call graphs
3. Usefulness of filters

Experiments conducted on 9 open source projects

2 Eclipse plugins, 2 SWT, 2 Swing, 3 Android applications

RESULTS

10 errors found in 9 programs (~ 90 KLOC), 5 new

2 false positives and 8 redundant warnings

Reflection-aware call graph found 2 bugs

CONCLUSION

Simple, yet elegant and efficient technique

Heuristics can be improved

More experiments needed

Subtle bugs to find with testing

```
private void deleteBucket() {
   public void run() {
     try { ... }
     catch (Exception e) {
       deleteError();
     }
   }
}

private void deleteError() {
   display.showErrorMessage("Delete failed");
}
```