

Synthesizing Program-Specific Static Analyses

Colin S. Gordon
Drexel University
OBT'18

Synthesizing Program-Specific Static Analyses

Colin S. Gordon
Drexel University
OBT'18

We Have Analyses for Many Properties

- Integrity
- Units of measure
- String formatting under concatenation, etc. (e.g., Regexp)
- Synchronization
- Nullability
- Confidentiality
- Taint Analysis
- Numerical range analysis
- ...

Why Target *Those* Problems?

- Designing a new type system is a lot of work
- Designing a new abstract domain is a lot of work
- Designing solvers for a new class of value constraints is a lot of work
- So we target properties that apply to *nearly every program*.
- This ensures that voluminous hard work pays dividends.

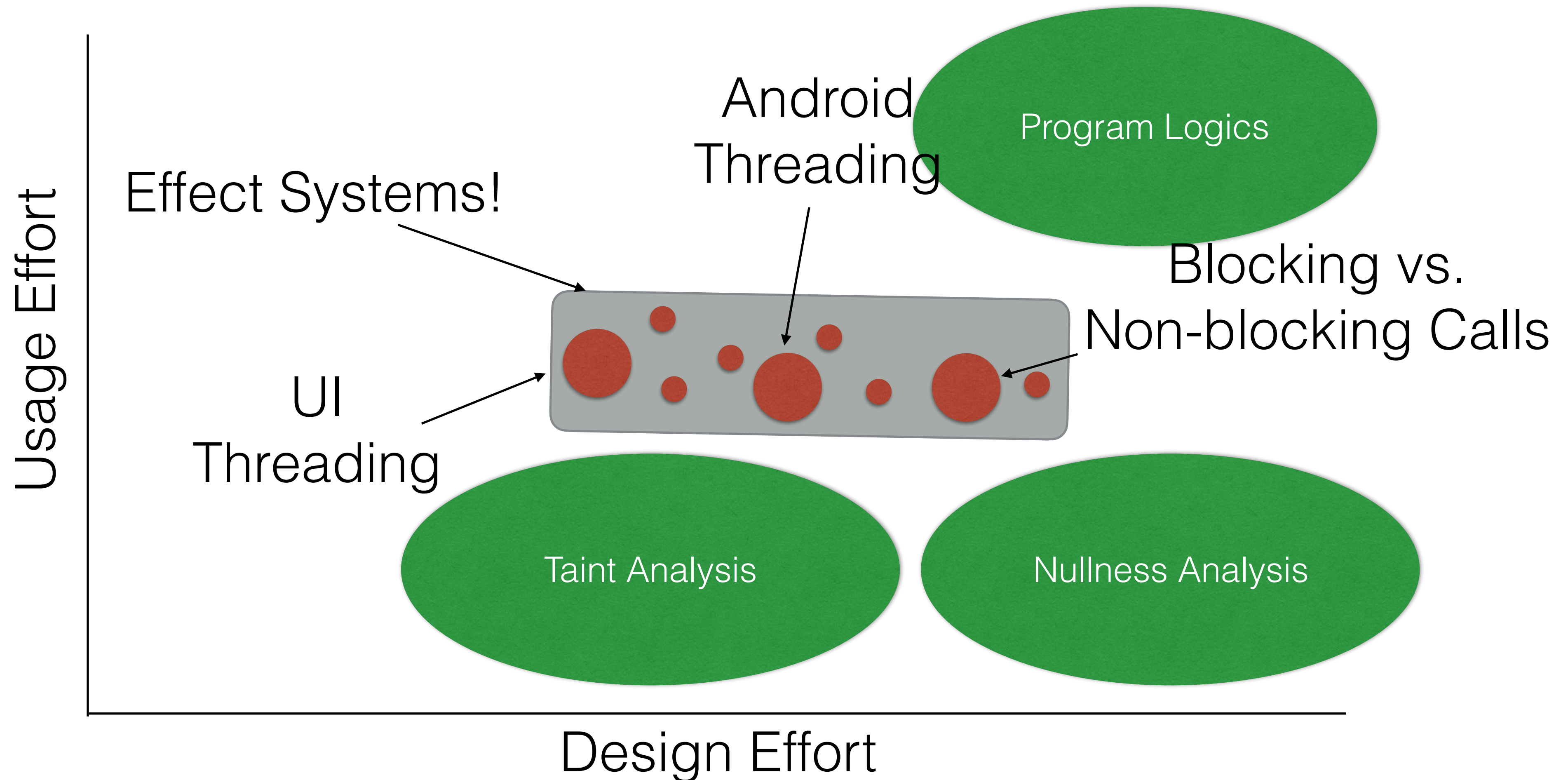
Broad Applicability

- Many programs benefit from checking nullness, taint, etc.
- Most programs also have *program-specific* correctness criteria
 - i.e., the specification for that program
- Currently, the only way to check those criteria is to use a general program logic or model checker
 - Lots of work to build, but widely applicable

Narrow Applicability

- Many programs would benefit from better checking of their program-specific correctness criteria
- Most development teams lack developers capable of using Iris, FCSL, etc., or debugging verification failure from VeriFast, Clousot, Dafny, Liquid Haskell, etc.
- Need something with “usability” of weaker systems, but ability to target program-specific criteria
- Most development teams lack expertise to design such systems

Design Effort vs. Usage Effort



Size is applicability

The Answer

- Synthesize analysis tools
 - Pick a constrained space (subset of abstract domains or pluggable type systems)
 - Developers provide positive or negative examples, in terms of problems they understand
 - A tool takes those examples, other constraints, and synthesizes a new static analysis / type system / etc.

Meeting in the Middle

- Type ...
 - Effect ...
 - Sequence ...
ordered monoid
 - Abstract Interpreters are parameterized by a complete lattice
 - ...
- If we can generate an appropriate partial order, we can generate a “usable” static analysis!
- We can teach undergrads to do this!
- Still requires program analysis expertise to model the problem.

Developers Don't See Lattices

- Various classes of systems correspond to different program-level constraints
 - Type Qualifiers restrict data flow
 - Effect Systems restrict the call graph

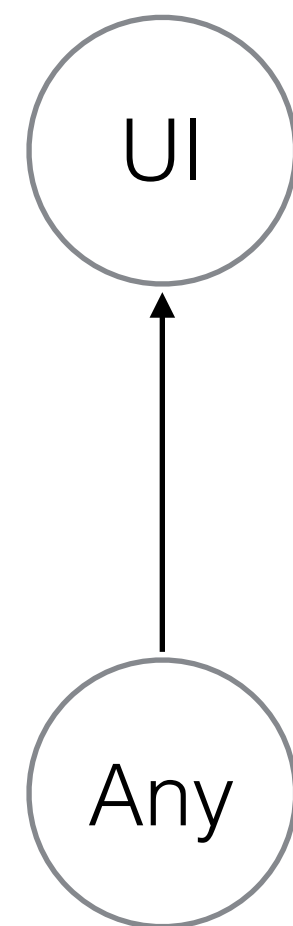
Developers understand data flow and call graphs!

Effects for Thread Confinement

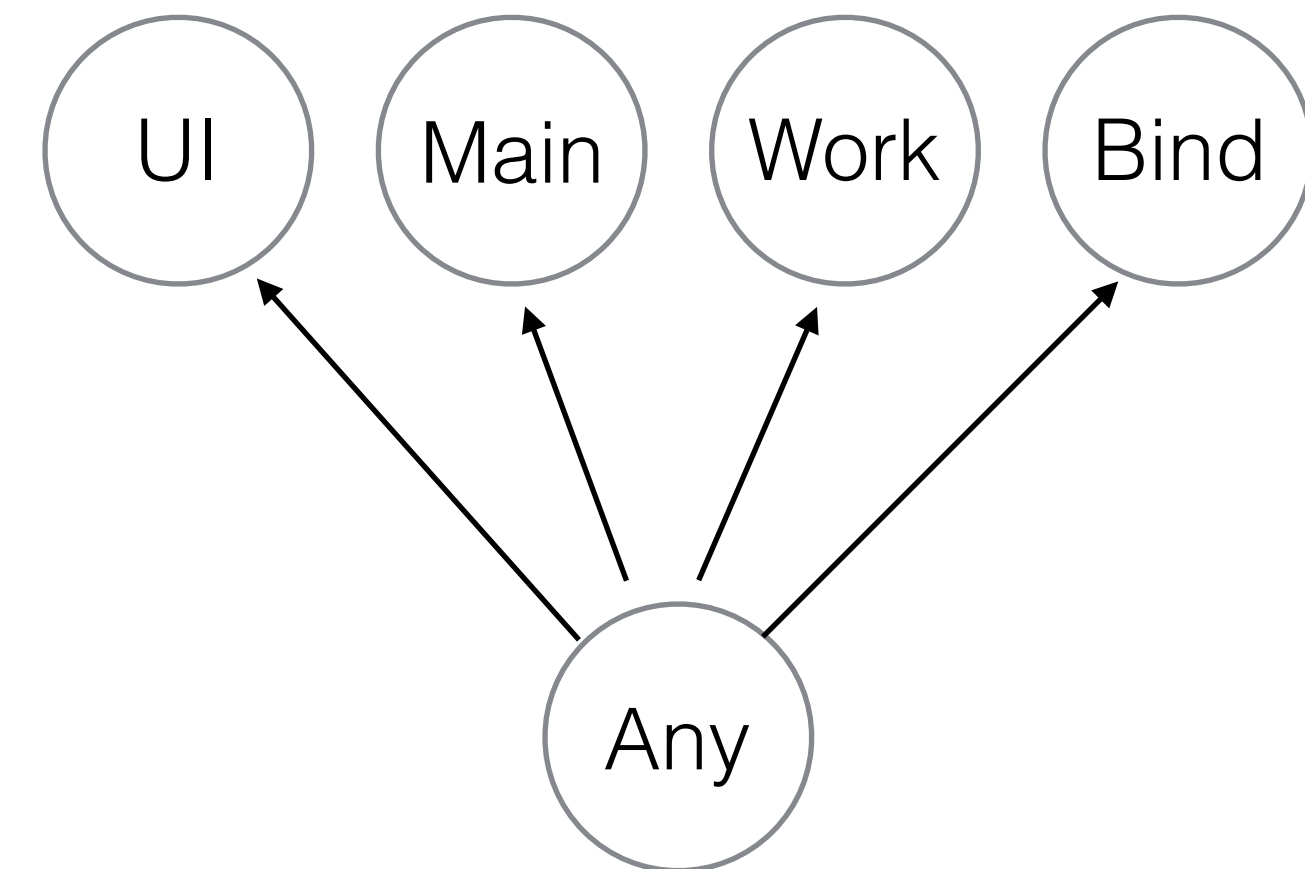
- Many runtime systems restrict certain actions to certain threads
 - e.g., all UI element updates on the UI event loop thread
 - e.g., all networking code on a dedicated thread
- Developers understand this. They just make mistakes.
- Correct thread confinement can be enforced via an effect system
- The choice of threads for effects *depends on the application*

Different Threading Models, Different Effects

Standard UI Library



Android UI Library



We should be able to synthesize these!

What If You Violate UI Thread Confinement?

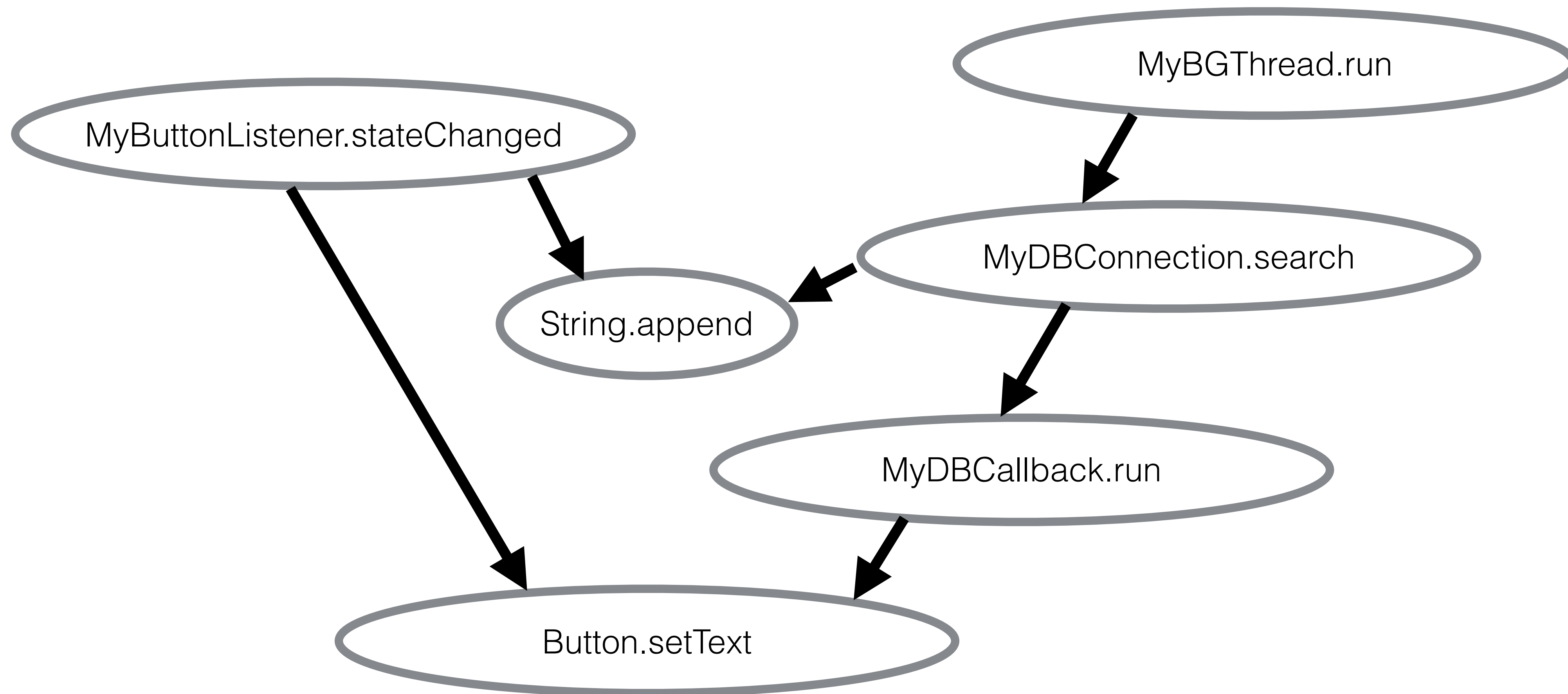
- An exception!
- With a stack trace!
- Which is a *counterexample*:
 - Gives a path in the call graph
 - At least one of these calls should be rejected by an effect system.
- Same thing happens on Android

```
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)
```

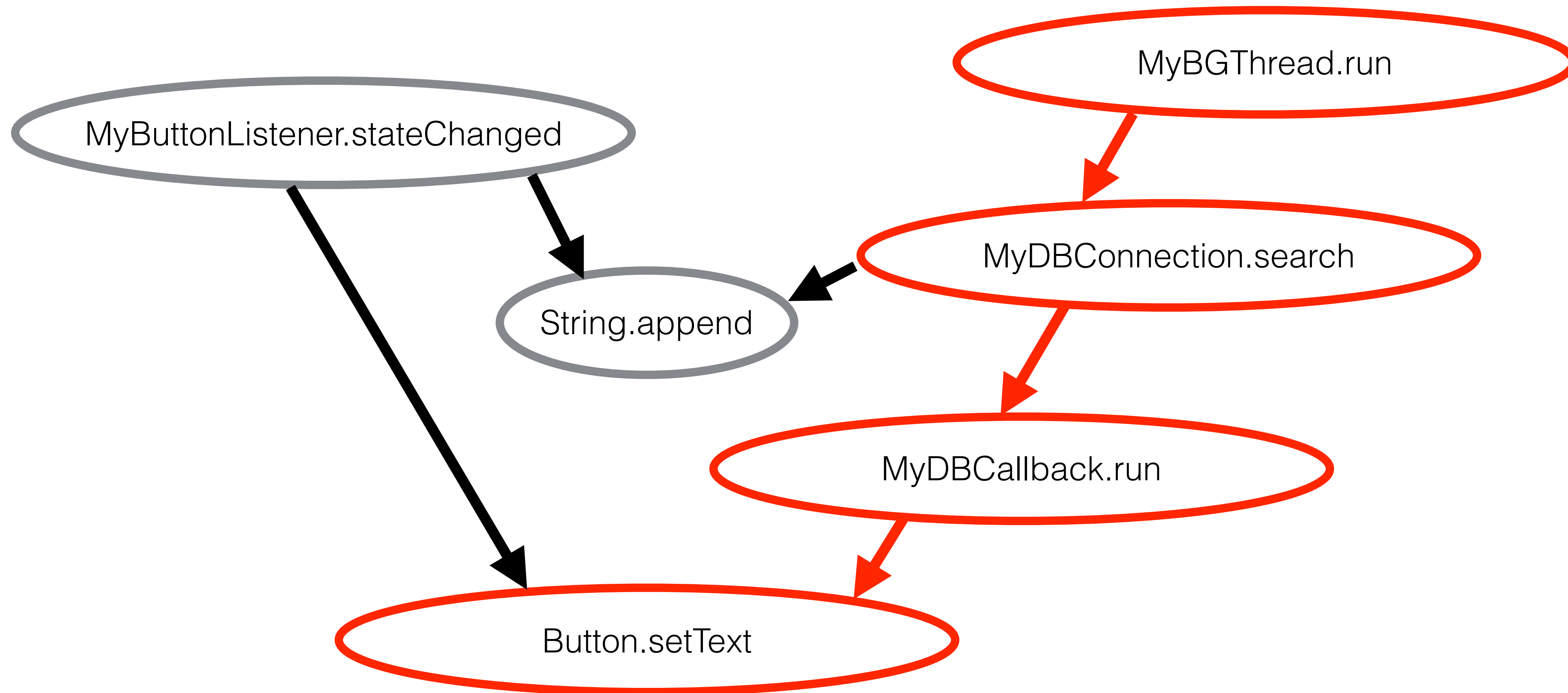
Synthesizing Effect Systems from Bad Call Examples

- The effect system for JavaUI (ECOOP'13, less polymorphism) could be synthesized:
 - Call-graph is a directed graph of executable elements (methods)
 - Effects correspond to regions of the graph
 - Subeffecting corresponds to uni-directional reachability b/t regions
 - Stack traces from exceptions identify a set from at least one edge must be rejected — a path between regions in the wrong direction
 - Call edges within UI libraries are “hard constraints”
 - Program edges are “soft constraints”
- Hypothesis: The JavaUI effect lattice corresponds the minimum partitioning that fixes *all* bad stacks

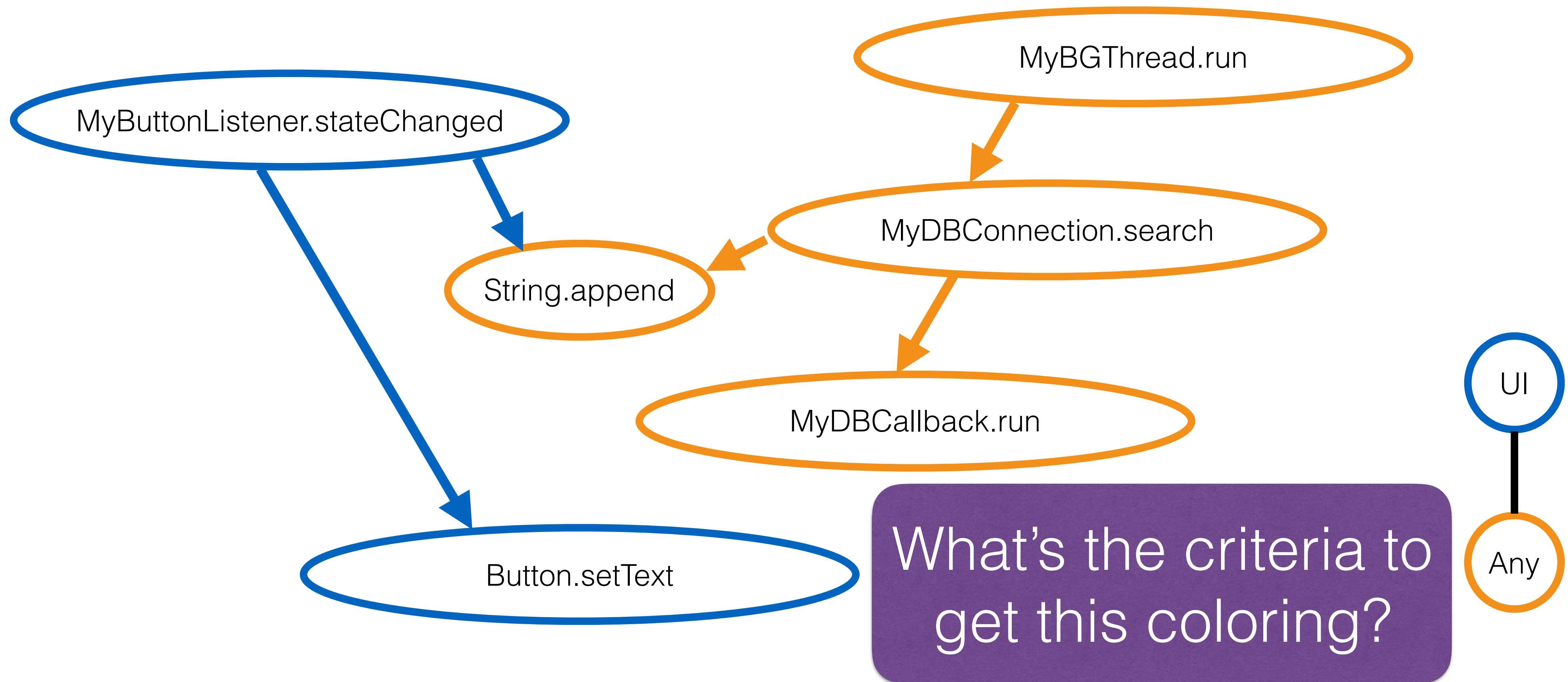
UI Call Graphs



UI Call Graphs



UI Call Graphs



Synthesizing More Effect Systems

Repeat for Android threading

Repeat for blocking calls

Synthesizing Taint Analysis

- Consider a two-qualifier taint type system (e.g., Shankar et al.'s “Detecting Format String Vulnerabilities with Type Qualifiers” USENIX Security 2001)
- Type qualifiers correspond to partitions of the data flow graph between storage locations
- Subtyping corresponds to uni-directional reachability between regions
- Minimal partitioning give you (monomorphic) taint qualifier system and library annotations

Challenges

- Polymorphism
- Informative error messages
 - Including when, e.g., a given problem doesn't map to effects/qualifiers/...
- Primitives & exceptions to the rule (`Display.syncExec(...)`, etc.)
- Getting enough examples to find the “right” system
- More sophisticated systems
 - Abstract domains are trickier than these, since they tend to be computationally expensive