Automated Testing with Targeted **Event Sequence** Generation

<u>Casper S. Jensen</u>
Aarhus University

Mukul R. Prasad
Fujitsu Laboratories of America

Anders Møller
Aarhus University

Lugano, ISSTA 2013

The Goal

- Automated testing of event-driven applications
  - Mobile applications
  - Web applications
  - Desktop/GUI applications
- Generation of event sequences
  - High code coverage

The Problem

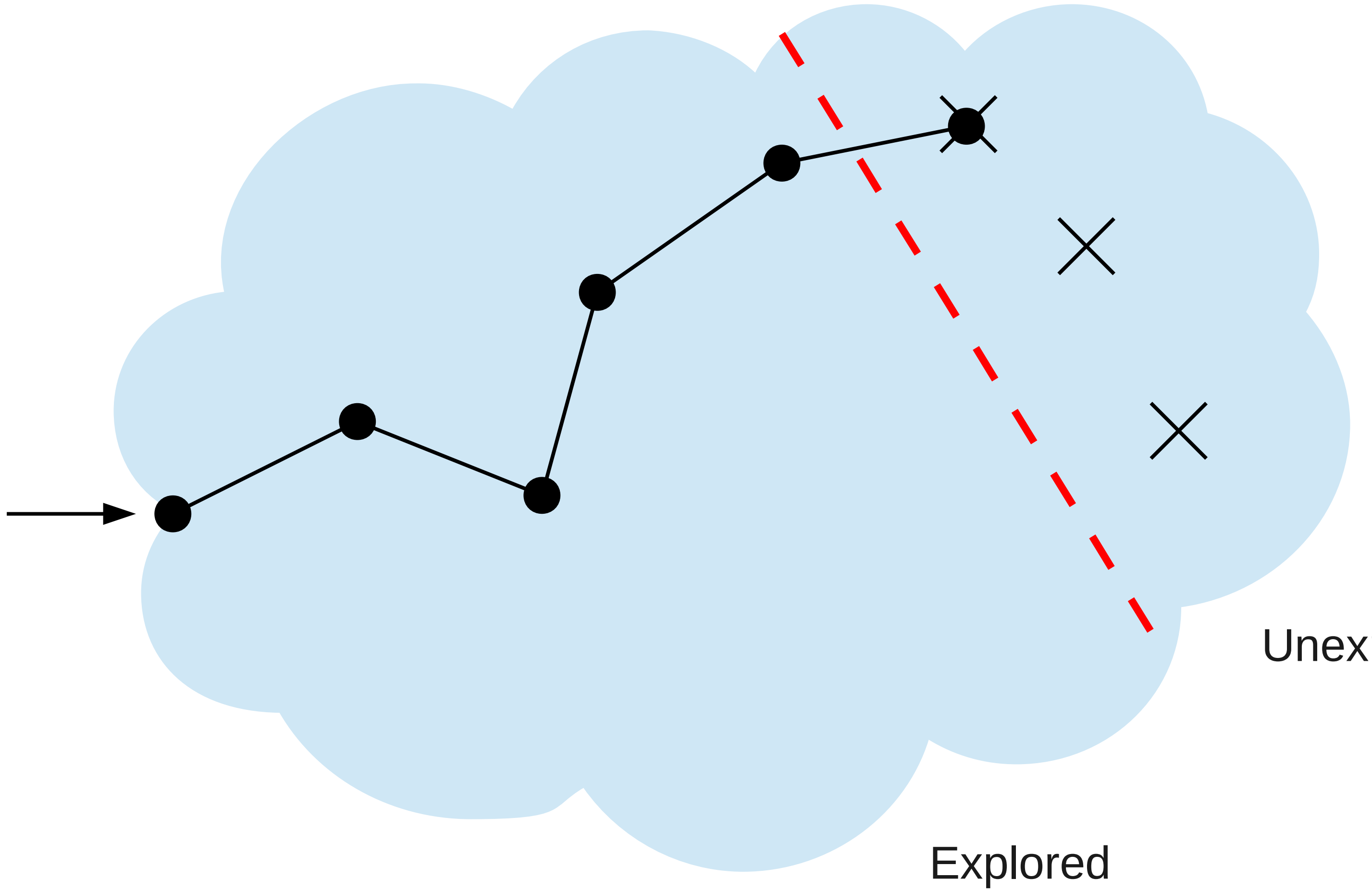
Example Event Sequence

1. Click button *foo*
2. Click button *bar*
3. **Input value “1” into field *foobar***
4. Press the back button
5. Click button *baz*
6. *Click button* “+”
7. *Click button* “+”

The Problem (cont.)

- Some branches require:
  - Long event sequences
  - Specific event parameters
- Promising methods:
  - *Random*, *feedback-directed*, *GUI-driven* and *fuzz-testing*

The Search Space



Unexplored

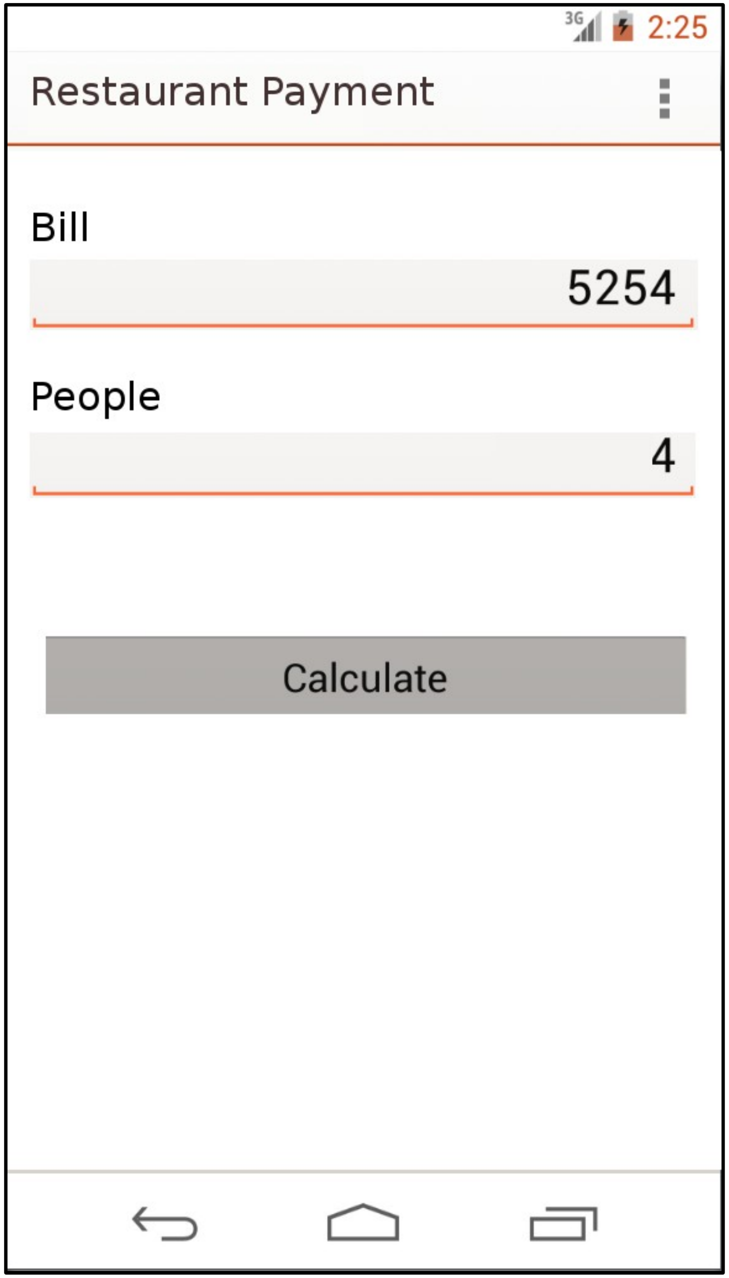
Explored

A **targeted** method for **event sequence** generation for **event-driven applications**

Outline

1. Motivating Example
2. Observations
3. Key Idea
4. Solution and Algorithm
5. Implementation
6. Evaluation
7. Conclusion

Motivating Example



Restaurant Payment

Bill

5254

People

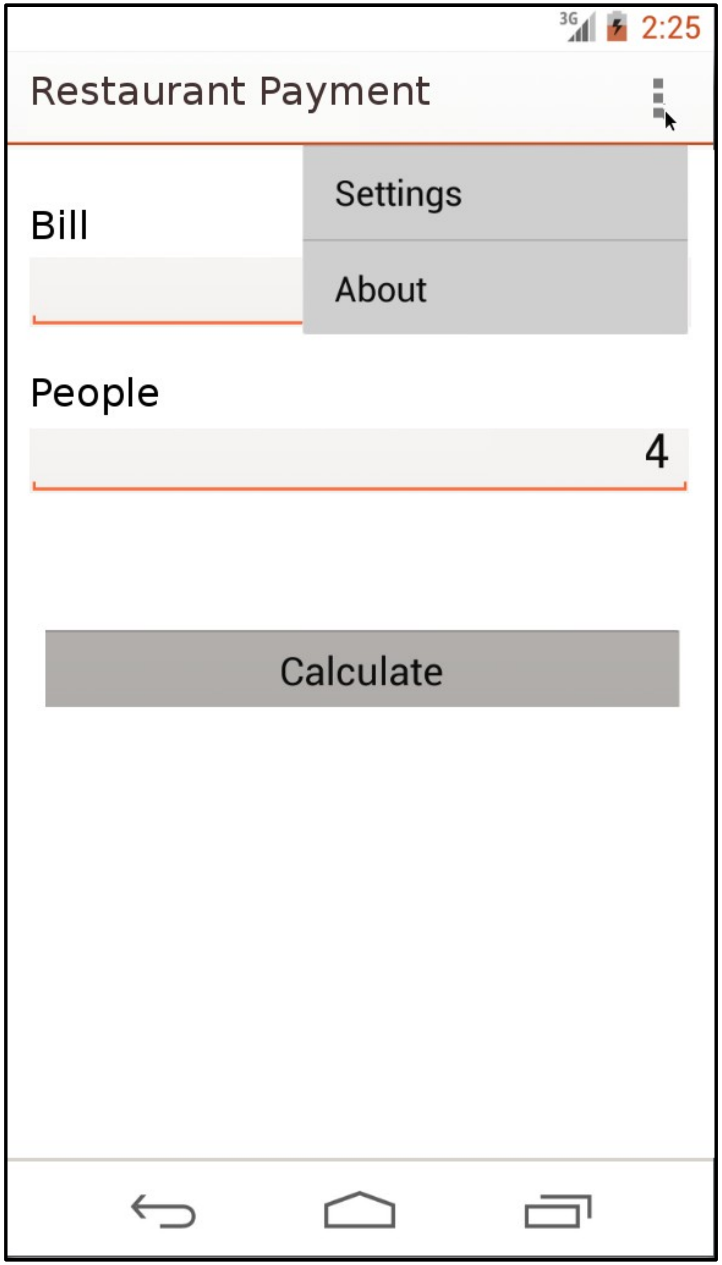
4

Restaurant Payment

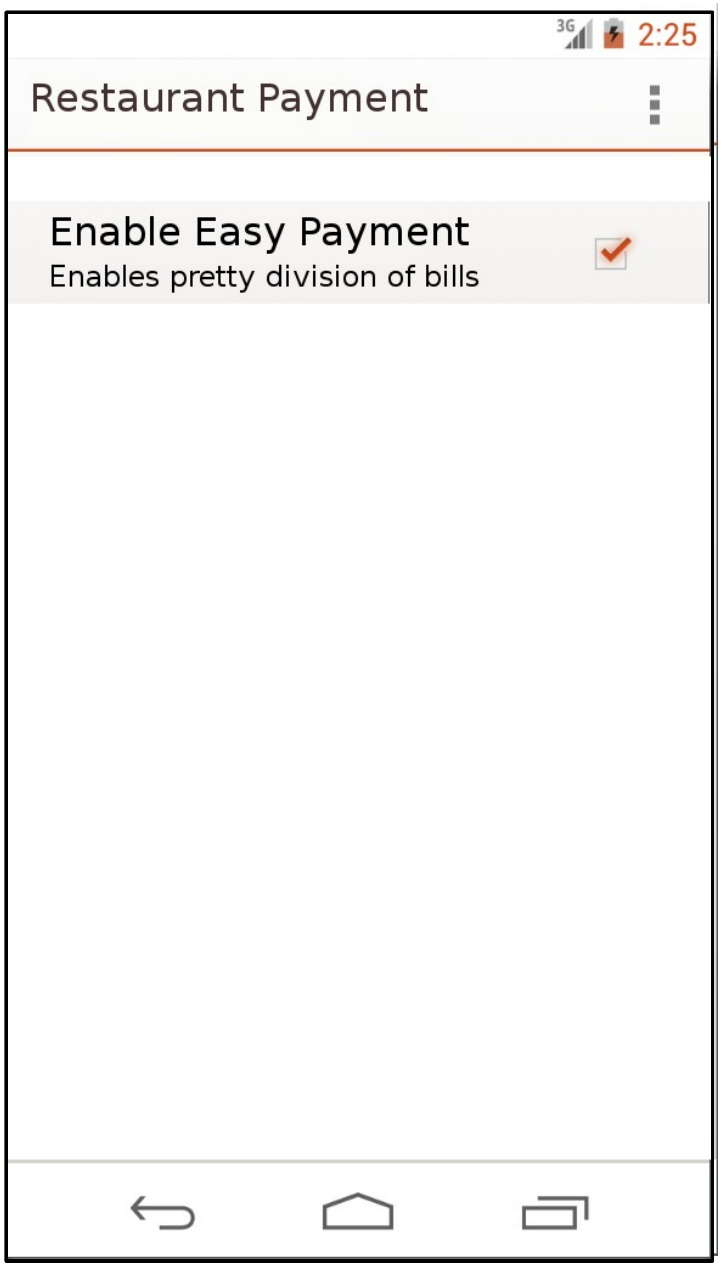
Divide restaurant payments between participants.

(Simplified version of one of our benchmarks)

Motivating Example



Motivating Example

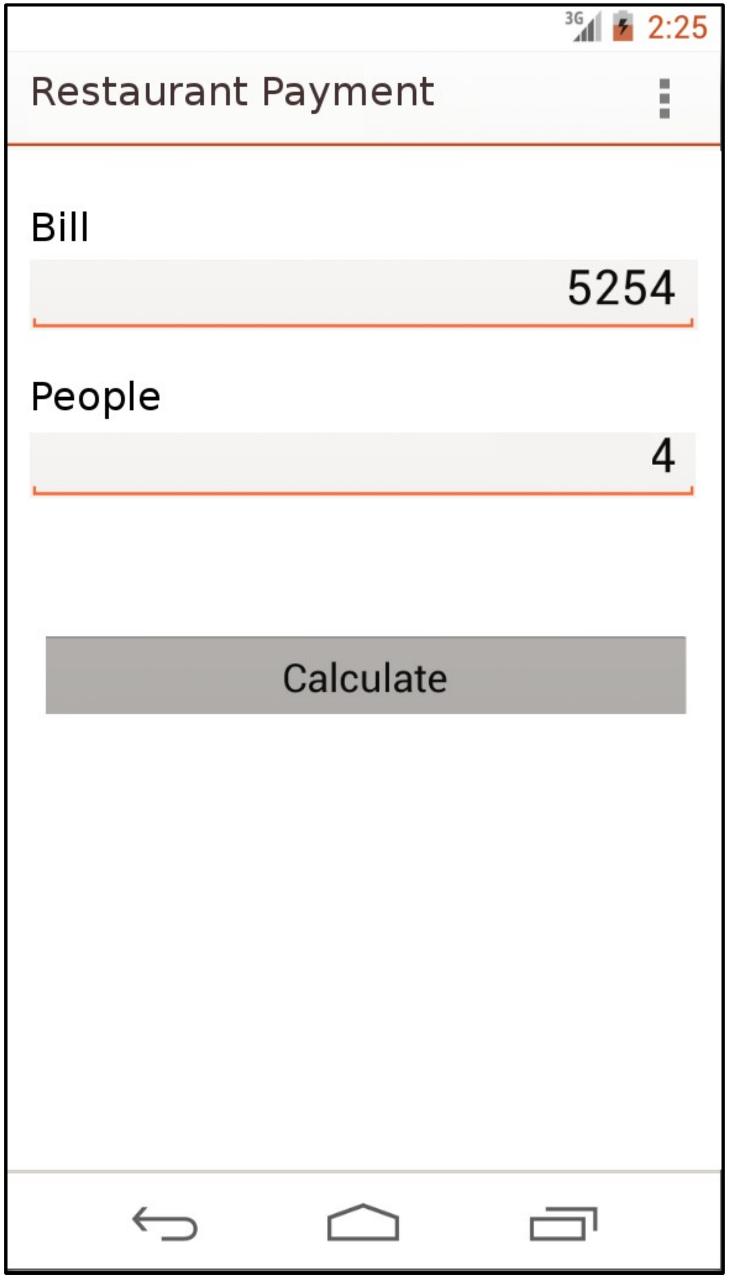


Restaurant Payment

Enable Easy Payment ☑

Enables pretty division of bills

Motivating Example



Restaurant Payment

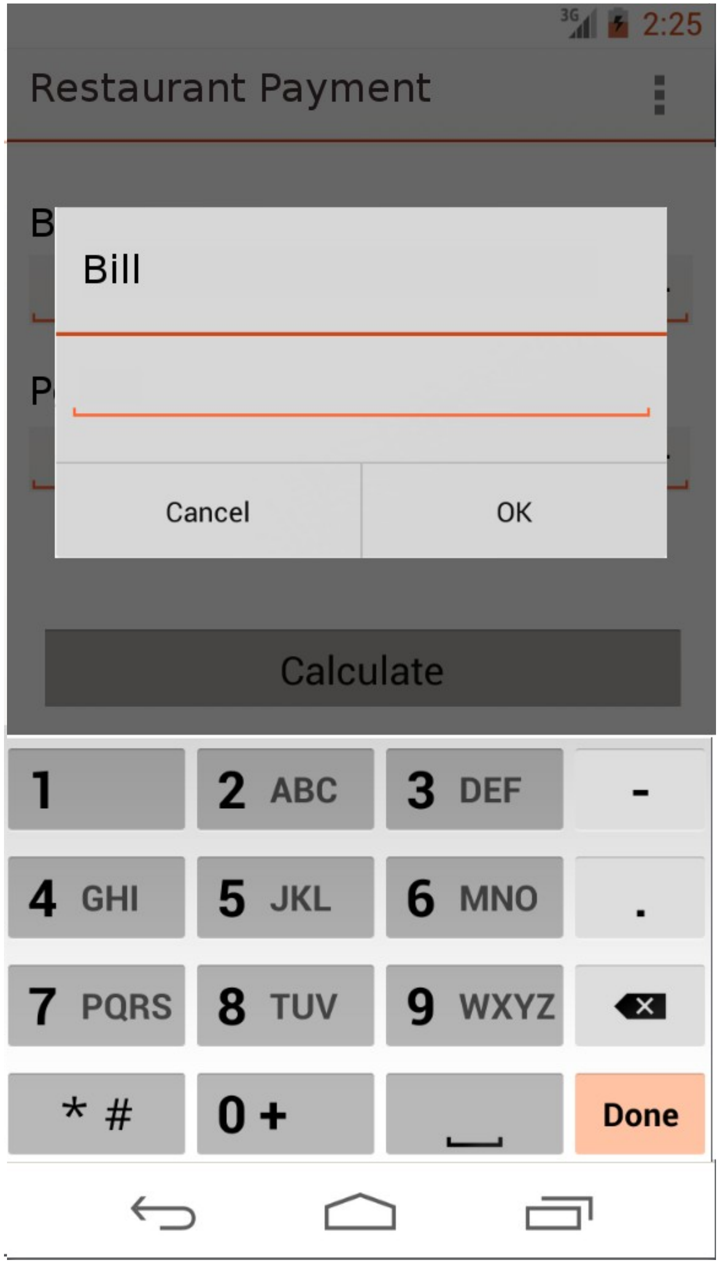
Bill

5254

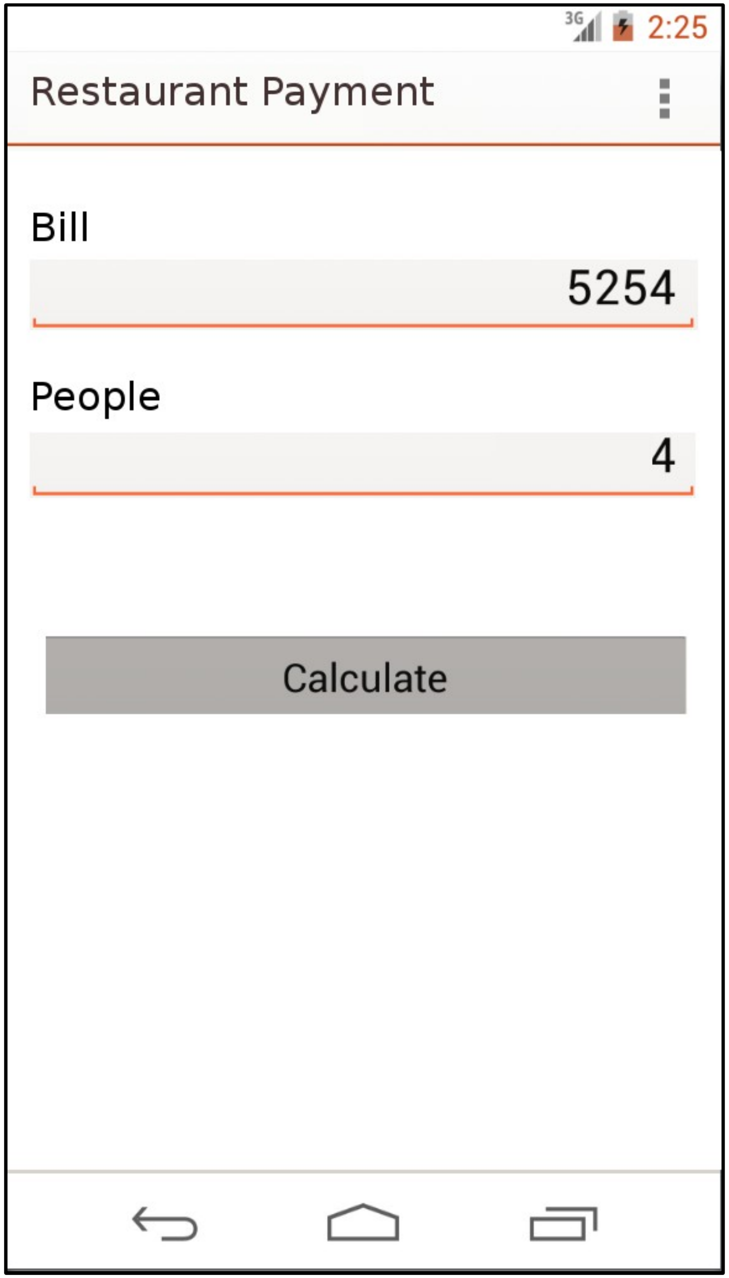
People

4

Motivating Example



Motivating Example



Restaurant Payment

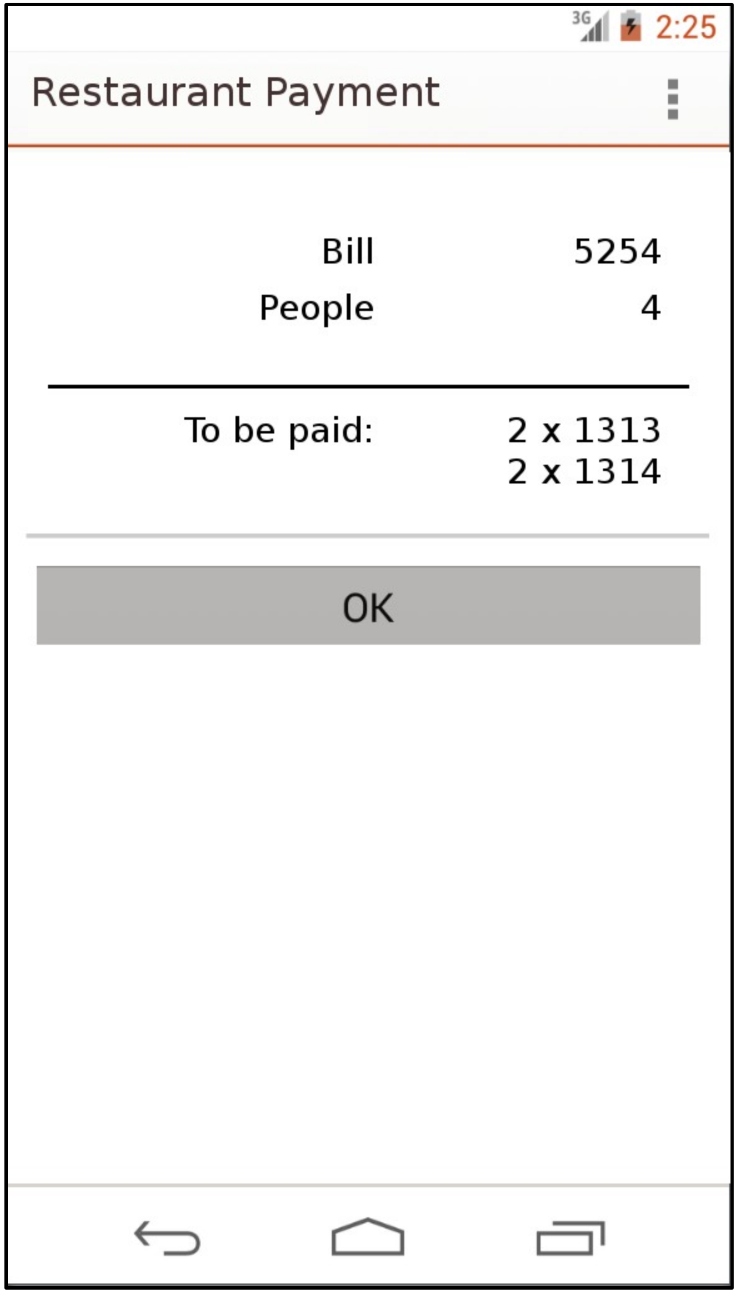
Bill

5254

People

4

Motivating Example



Bill	5254
People	4
To be paid:	2 x 1313 2 x 1314

The actual calculation

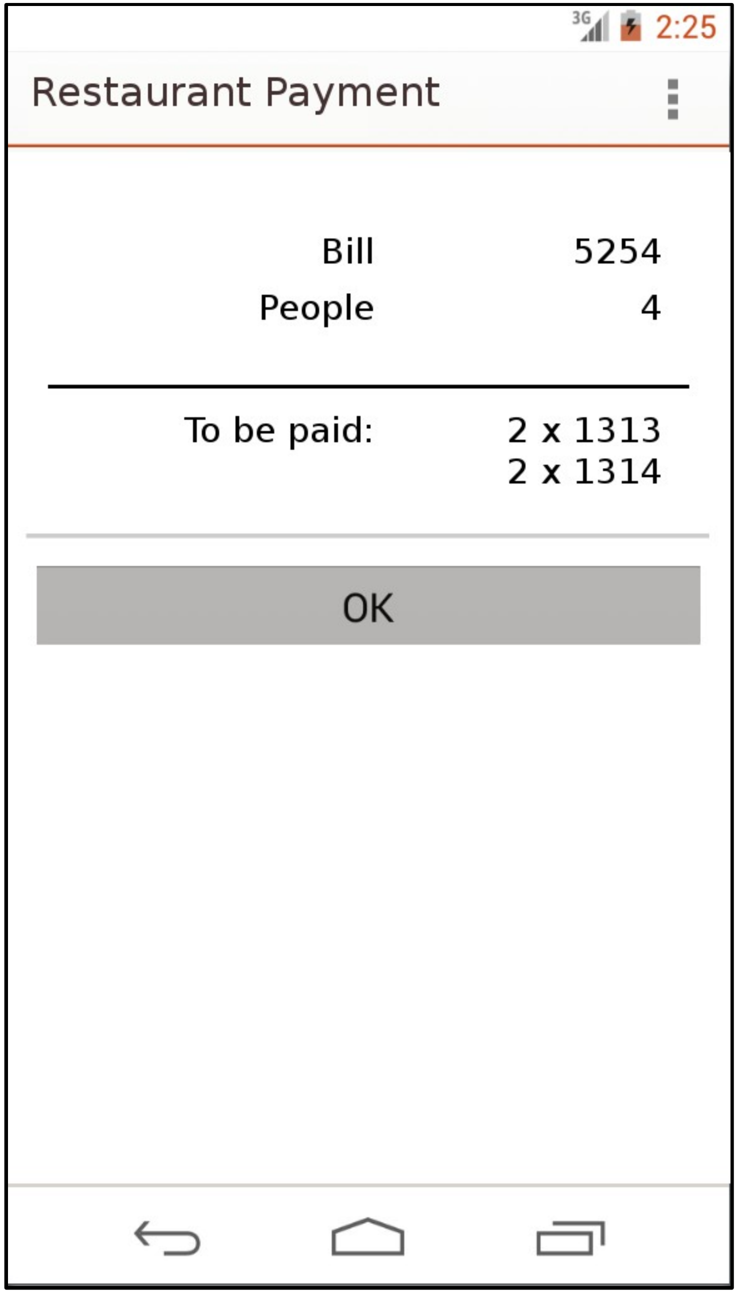
```
int bill = this.appState.enteredAmount;
int people = this.appState.numberOfPeople;
boolean useEasyPay = this.appState.numberOfPeople;

if (useEasyPay) {
    ...

    if (bill < people) {
        ... // interesting calculations
        return
    }

    ...
```

Motivating Example



Restaurant Payment

Bill	5254
People	4
To be paid:	2 x 1313 2 x 1314

The actual calculation

```
int bill = this.appState.enteredAmount;
int people = this.appState.numberOfPeople;
boolean useEasyPay = this.appState.numberOfPeople;

if (useEasyPay) {
    ...

    if (bill < people) {
        ... // interesting calculations
        return
    }

    ...
```

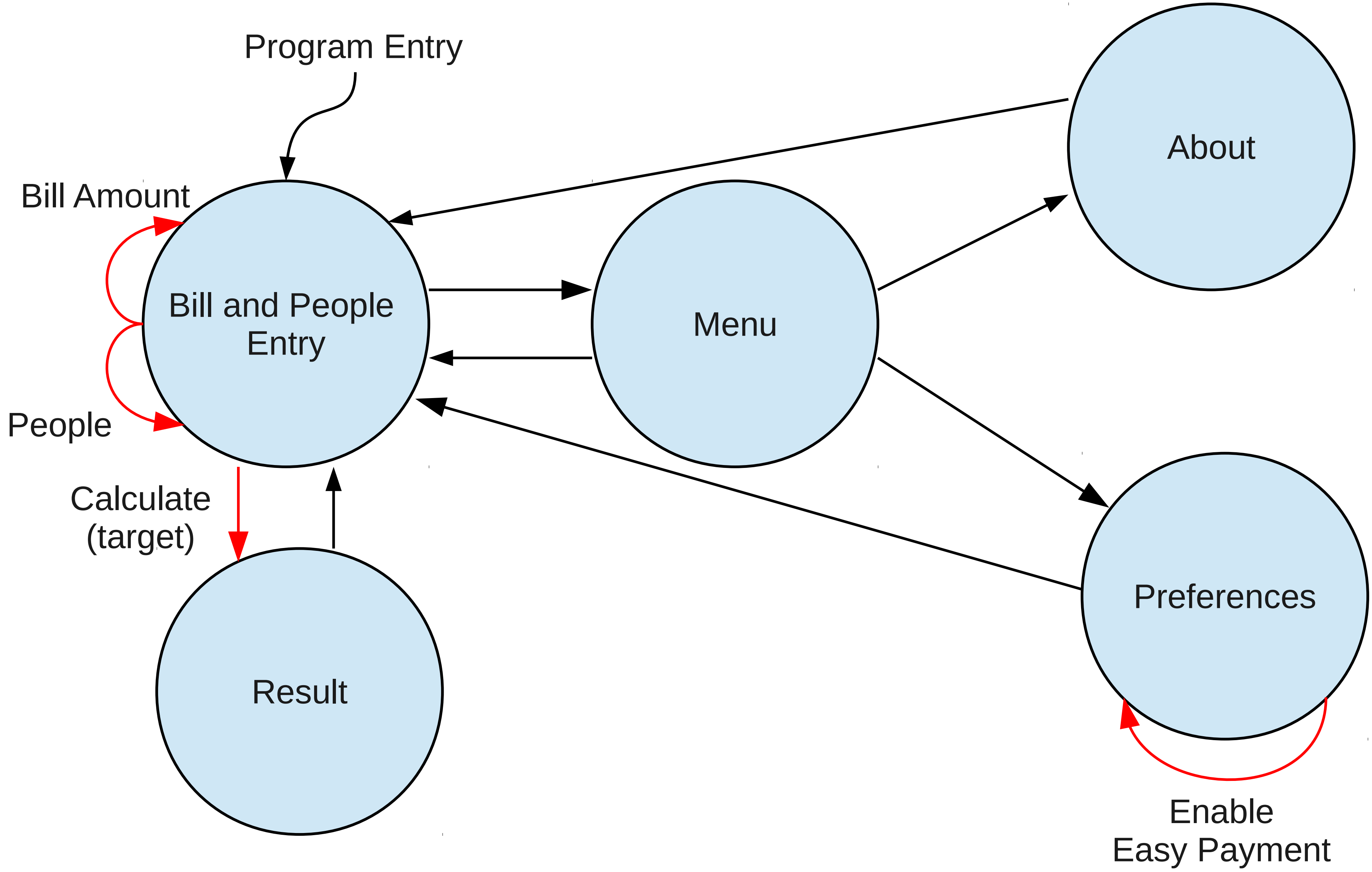
`useEasyPay ∧ bill < people`

Observation 1

- A branch (target) **depends** on program state
- Some events (“anchor events”) **mutate** relevant program state

- The relevant program state is mutated at a limited number of program points
  - Only focusing on these points reduces the search space

UI Model


Program Entry
Bill Amount
People
Bill and People Entry
Menu
About
Calculate (target)
Result
Preferences
Enable Easy Payment

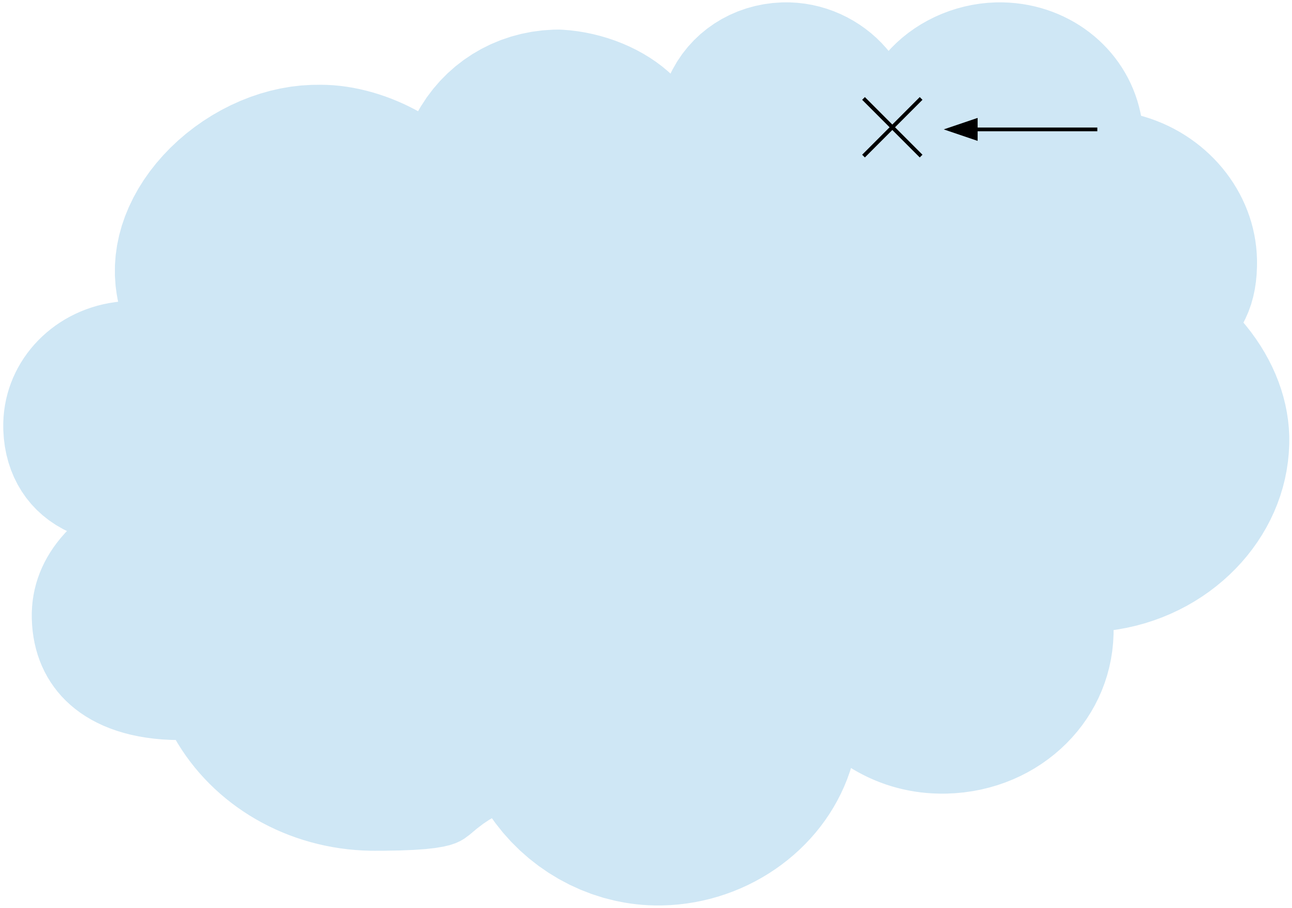
Observation 2

- **Anchor events** connected by **connector events**
- Program state is unaffected by connector events
  - The program state dependent on by the target
- Any feasible path between anchors will do

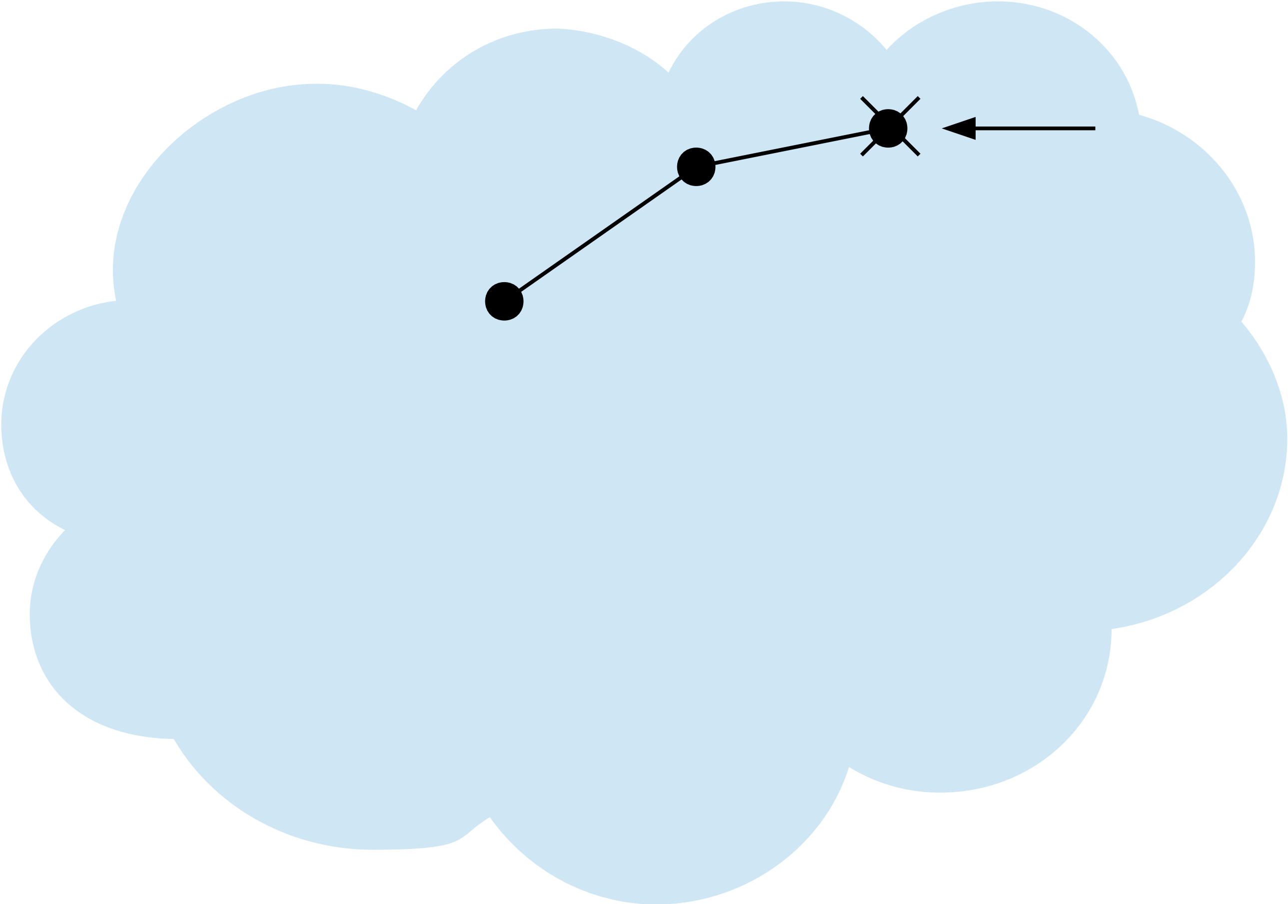
Key Idea

- For a **target branch**:
  - identify a sequence of **anchors**
  - changing **program state**
  - satisfying the target branch's dependencies
- Focus on *anchors*, omitting unrelated parts of the application
- **Discard** infeasible sequences as early as possible

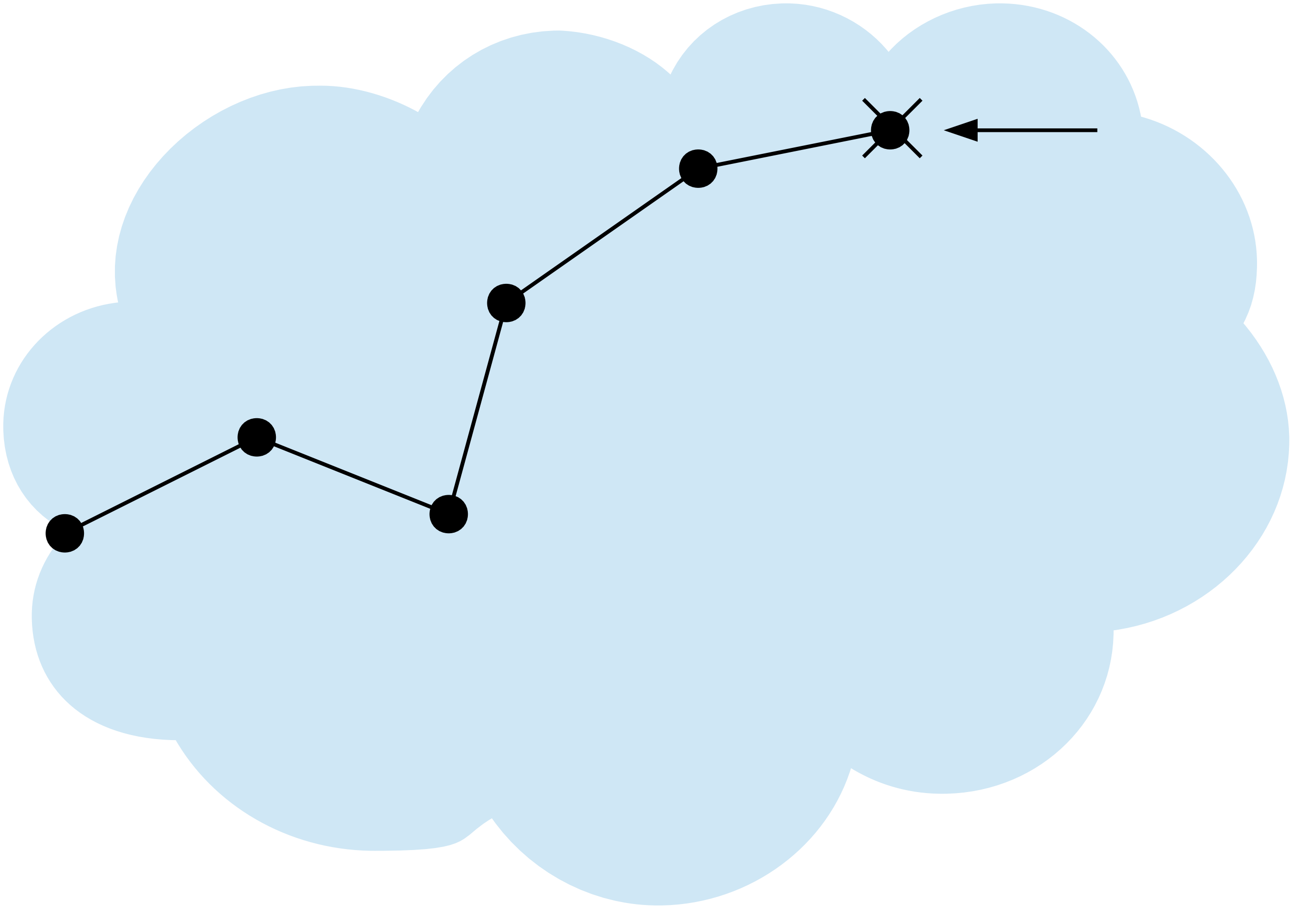
The Search Space



The Search Space



The Search Space

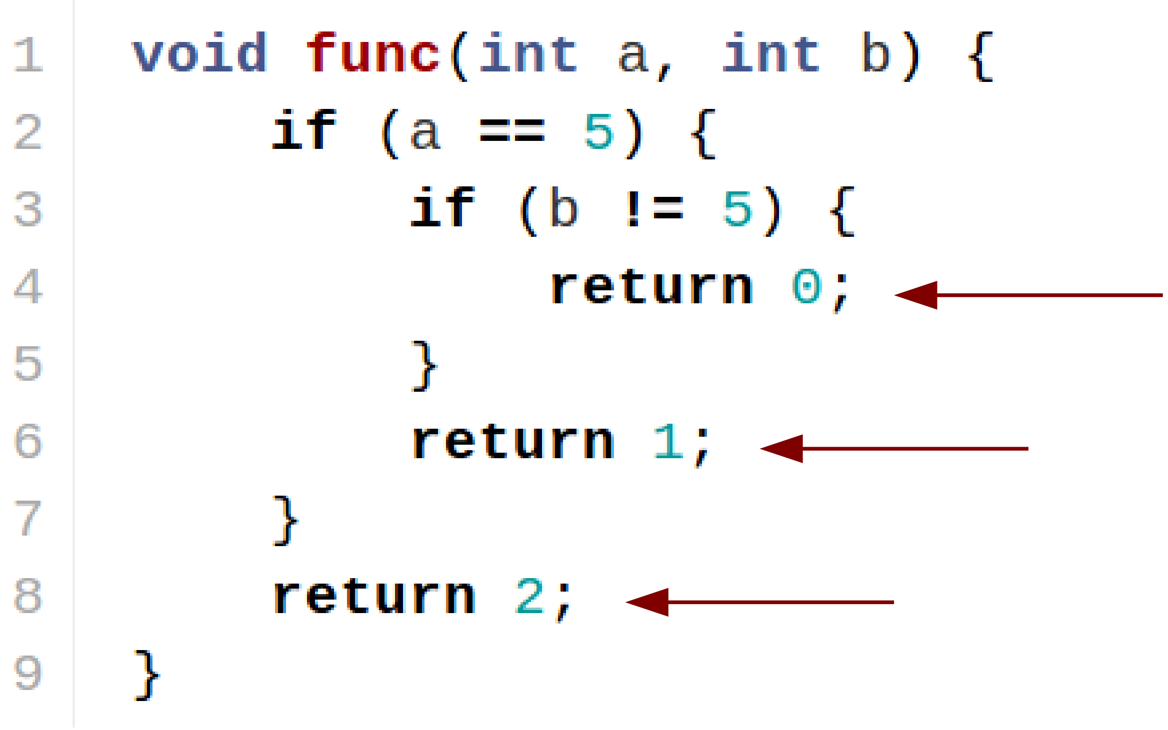


Our Solution

Targeted Sequence Generation
using
UI Models and Concolic Testing

Recall: Concolic Testing

- Symbolic and Concrete execution
- Explore the execution space of a program
  - Uses a constraint solver



```
void func(int a, int b) {
    if (a == 5) {
        if (b != 5) {
            return 0;
        }
        return 1;
    }
    return 2;
}
```

$a = 5 \land b \neq 5$

$a = 5 \land b = 5$

$a \neq 5$

Our High-level Algorithm

(1) Summarization Phase

Run concolic testing on each event handler

- Path conditions for individual program traces
- Dependencies / program mutation

(2) Sequence Generation Phase (repeat for each target)

Backward search, starting with a *target branch* and backward to an entry point

- Uses UI models, path conditions, dependencies

Implementation

Collider

Automated testing of Android mobile applications

- Implemented using:
  - Symbolic-JPF (solver infrastructure)
  - The Android Emulator (concrete execution)

- Targets the Dalvik bytecode level
  - only a compiled executable is required for testing

Evaluation

- Study on 5 Android applications
- Random testing (Monkey) and UI-driven testing is used to reach a number of targets
- Of 99 unreached targets:
  - **Collider reaches 46 of these**
  - Remaining unreached mainly due to limitations in the constraint solver
  - Phase 1: 3 to 5 hours
    Phase 2: 2 seconds to 30 minutes

Conclusion

- A new method for targeted *event sequence* generation
- Search is guided through *concolic testing* and *UI-models*
- Implementation and evaluation shows the potential of the method
- Generation of sequences of events for reaching highly constrained targets