Systematic Execution of Android Test Suites in Adverse Conditions

Christoffer Quist Adamsen, Gianluca Mezzetti, Anders Møller
Aarhus University, Denmark

ISSTA 2015, Baltimore, Maryland

Motivation

- Mobile apps are difficult to test thoroughly
- Fully automated testing tools:
  - capable of exploring the state space systematically
  - no knowledge of the intended behaviour
- Manually written test suites widely used in practice
  - app largely remains untested in presence of common events

Goal

Improve manual testing under adverse conditions

1. Increase bug detection as much as possible
2. Run test suite without significant slowdown
3. Provide precise error messages

Methodology for testing

- Systematically expose each test to adverse conditions, where unexpected events may occur during execution
- Which unexpected events does it make sense to systematically inject?

Neutral event sequences

- An event sequence *n* is **neutral** if injecting *n* during a test *t* is not expected to affect the outcome of *t*
- We suggest a general collection of useful neutral event sequences that e.g. stress the life-cycle of Android apps
  - Pause → Resume
  - Pause → Stop → Restart
  - Pause → Stop → Destroy → Create
  - Audio focus loss → Audio focus gain
  - …

Example

Injection points

```
public void testDeleteCurrentProject() {
  createProjects();
  c
  l                                     CT);
  clickOnText("Delete");
  clickOnText("Yes");
  assertFalse("project still visible",
              searchText(DEFAULT_PROJECT);
  …
}
```

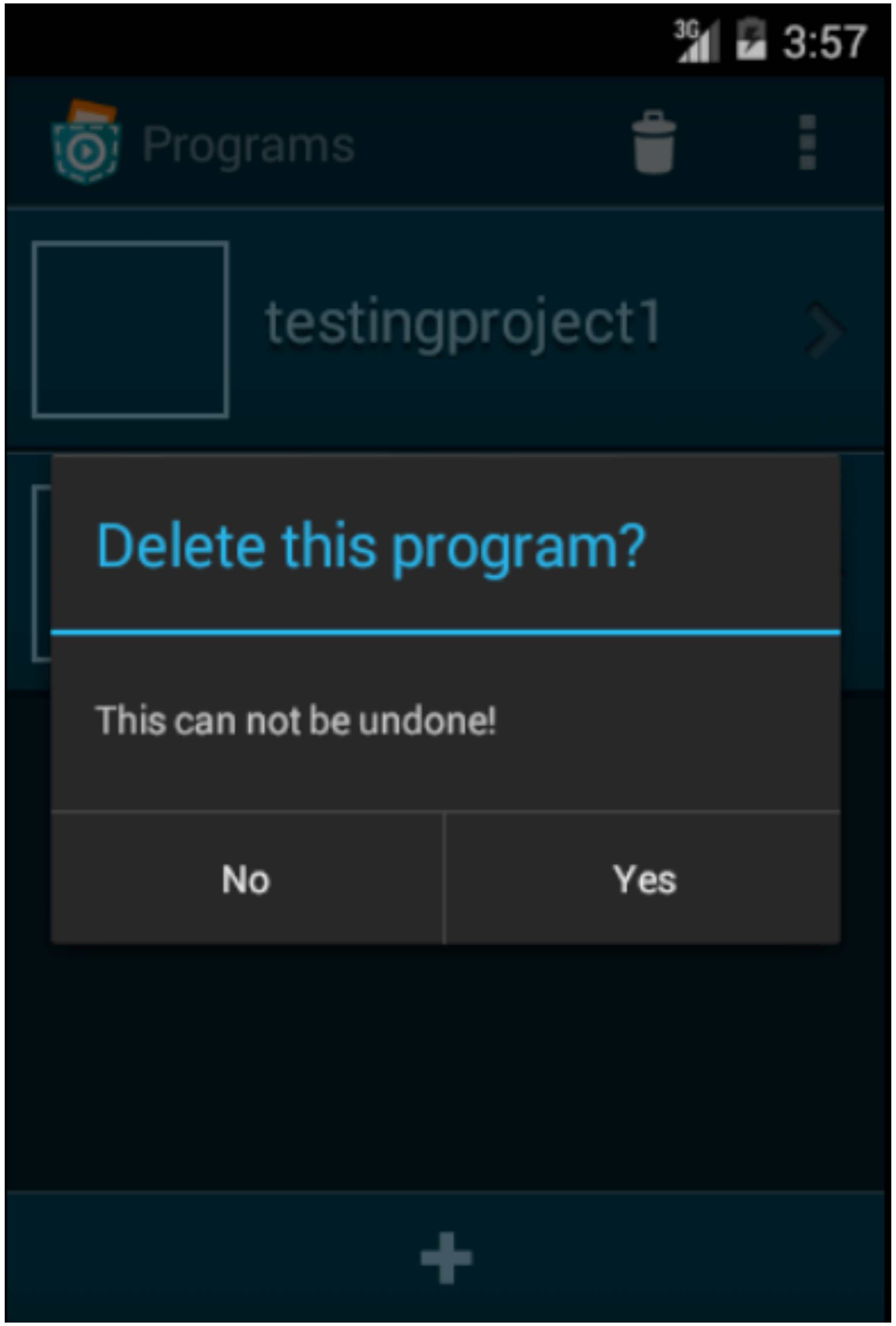
Execute each neutral event sequence at each injection point

Example

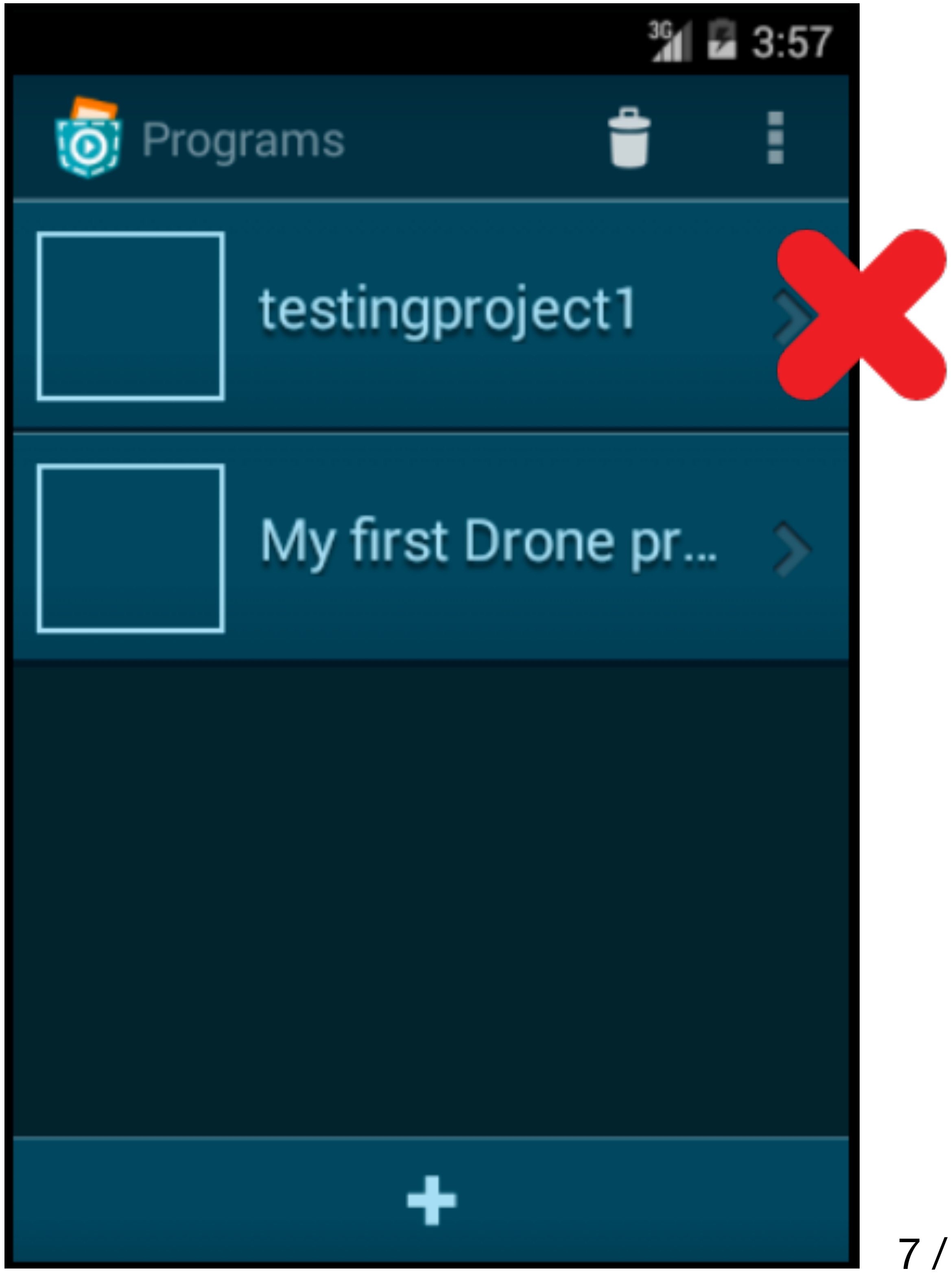
Injection points

```
public void testDeleteCurrentProject() {
    createProjects();
    clickOnButton("Programs");
    longClickOnTextInList(DEFAULT_PROJECT);
    clickOnText("Delete");
    clickOnText("Yes");
    assertFalse("project still visible",
                searchText(DEFAULT_PROJECT));
    …
}
```

Example



3:57
Programs
testingproject1
Delete this program?
This can not be undone!
No
Yes



3:57
Programs
testingproject1
My first Drone pr...

Example

```
public void testDeleteCurrentProject() {
    createProjects();
    clickOnButton("Programs");
    longClickOnTextInList(DEFAULT_
    clickOnText("Delete");
    clickOnText("Yes");
    assertFalse("project still visible",
                    searchText(DEFAULT_PROJECT);
    …
}
```

Injection points

Strategy may be too aggressive

Hypothesis for aggressive injection strategy

Few additional errors will be detected by:

- injecting a subset of the neutral event sequences, and
- using only a subset of the injection points

Example

Injection points

```
public void testDeleteCurrentProject() {
    createProjects();
    clickOnButton("Programs");
    longClickOnTextInList(DEFAULT_
    clickOnText("Delete");
    clickOnText("Yes");
    assertFalse("project still visible",
                searchText(DEFAULT_PROJECT);
    ... ...
}
```

Failure potentially shadows others

Evaluating the error detection capabilities

- Empirical study using our implementation **Thor** on 4 open-source Android apps (with a total of 507 tests)
- To what extent is it possible to trigger failures in existing test suites by injecting unexpected events?
- 429 tests of a total of 507 fail in adverse conditions!
- 1770 test failures counted as distinct failing assertions (none of which appear during ordinary test execution)

Evaluating the error detection capabilities

- Manual classification of 682 of the 1770 test failures revealed 66 distinct problems

App	Logical					UI	
	Crash	Silent fail	Not persisted	User setting lost	…	Element disappears	…
Pocket Code	1 (9)	7 (42)		1 (6)		14 (104)	
Pocket Paint	2 (45)		1 (4)	4 (42)		9 (131)	
Car Cast	1 (7)					5 (18)	
AnyMemo						4 (15)	

#distinct problems (#error messages)

Evaluating the error detection capabilities

Only 4 of 22 distinct bugs that damage the user experience are crashes

- …est failures

	Logical					UI	
App	Crash	Silent fail	Not persisted	User setting lost		Element disappears	
Pocket Code	1 (9)	7 (42)		1 (6)		14 (104)	
Pocket Paint	2 (45)		1 (4)	4 (42)		9 (131)	
Car Cast	1 (7)				…	5 (18)	…
AnyMemo						4 (15)	

Evaluating the error detection capabilities

- Manual classification of 682 of the 17
revealed 66 distinct problems

Failures dominated by UI glitches

	Logical					UI	
App	Crash	Silent fail	Not persisted	User setting lost		Element disappears	
Pocket Code	1 (9)	7 (42)		1 (6)		14 (104)	
Pocket Paint	2 (45)		1 (4)	4 (42)		9 (131)	
Car Cast	1 (7)				…	5 (18)	…
AnyMemo						4 (15)	

Evaluating the execution time

- Competitive to ordinary test executions

	App			
Strategy	AnyMemo	Car Cast	Pocket Code	Pocket Paint
Basic	1.05x	1.21x	1.38x	0.99x

Evaluating the execution time

- Competitive to ordinary test executions

	App			
Strategy	AnyMemo	Car Cast	Pocket Code	Pocket Paint
Basic	1.05x	1.21x	1.38x	0.99x
Rerun	2.11x	3.09x	4.70x	3.70x

Summary of evaluation

- Successfully increases the error detection capabilities!
- App crashes are only the tip of the iceberg
- Small overhead when not rerunning tests

Goal, revisited

Improve manual testing under adverse conditions

1. Increase bug detection as much as possible
2. **Run test suite without significant slowdown**
3. Provide precise error messages

Problems with rerunning tests

- Rerunning tests to identify additional bugs is expensive
- More assertion failures or app crashes
  do not necessarily reveal any additional bugs
- For example, the following tests from Pocket Code
  check similar use cases to `testDeleteCurrentProject()`:
  - `testDeleteProject()`
  - `testDeleteProjectViaActionBar()`
  - `testDeleteProjectsWithSpecialChars()`
  - `testDeleteStandardProject()`
  - `testDeleteAllProjects()`
  - `testDeleteManyProjects()`

Heuristic for reducing redundancy

- During test execution, build a cache of abstract states
- Omit injecting *n* in abstract state s after event e,
  if (n, s, e) already appears in the cache

Evaluating the redundancy reduction

- The redundancy reduction improves performance and results in fewer duplicate error messages!
- Case study on Pocket Paint:
  - Execution time reduces from 2h 48m to 1h 32m
  - 79% less error messages
  - 14 of the 17 distinct problems spotted

Goal, revisited

Improve manual testing under adverse conditions

1. Increase bug detection as much as possible
2. Run test suite without significant slowdown
3. **Provide precise error messages**

Isolating the causes of failures

- Since multiple injections are performed in each test, it may be unclear which injection causes the failure

Hypothesis for failure isolation

Most errors can be found by:

- injecting only one neutral event sequence, and
- using only one injection point

Isolating the causes of failures

For failing tests, apply a simple variant of delta debugging:

1. **Identify a neutral event sequence *n* to blame**
   Do a binary search on the neutral event sequences (keeping the injection points fixed)

2. **Identify the injection point to blame**
   Do a binary search on the sequence of injection points (injecting only *n*)

Evaluating the failure isolation

Failure isolation works!

- Applied the failure isolation to all 429 failing tests
- Successfully blamed a single neutral event sequence and injection point for **all 429 except 5 failures**

Conclusion

- Light-weight methodology for improving
  the bug detection capabilities of existing test suites
- Key idea: Systematically inject neutral event sequences
- Evaluation shows:
  - can detect many app-specific bugs
  - small overhead
  - precise error messages
- http://brics.dk/thor