Fully Automatic and Precise Detection of Thread Safety Violations

Authors:
Michael Pradel & Thomas R. Gross
2012
Motivation

• Generally, writing software is **fun**
• Coding (unit) tests however is **boring**
• Writing concurrent programs is challenging
• Writing effective tests that reveal concurrency bugs is even more challenging
Minimal effort required

• **Input:**
  – The class under test (CUT)
  – (Optional) Auxiliary classes and libraries that the CUT depends on

• **Output:**
  – True, non-redundant, concurrency bug reports
Method Overview

1. Generate concurrent test
2. Execute test repeatedly
3. Check whether thread safety violation caused a test to fail
4. Go back go #1
Test generation

• Goal: generate tests likely to expose concurrency bugs
  → Let the threads share state
• Method: split each test into a prefix $p$ and suffixes $s_1, \ldots, s_n$
  – The prefix creates an instance of the CUT and then “grows” it by calling its methods
  – The suffixes make further calls to the same CUT instance
Test generation

• To instantiate the CUT, the generator randomly selects a method that has the CUT as a return type.
  – This includes the constructor of the class.
  – If this method requires parameters, it will attempt to generate them automatically.

• Random CUT methods are then selected to “grow” and test the CUT instance.
  – A field access may also be selected.
  – Return values from method calls are stored in variables, which may be used as parameter values for future calls.

• Code sequences which, when run sequentially, result in an exception are discarded.
Thread safety

• Difficult to prove, easier to disprove.
  – We just need to find a counter-example
• Thread safety is a fuzzy term, many definitions
• The one adopted by the authors:
  – “A class is said to be **thread-safe** if multiple threads can use it **without synchronization** and if the behavior observed by each thread is **equivalent** to a **linearization** of all calls that **maintain the order** of calls in each thread”
Equivalent executions

• Authors’ definition:
  – Two executions $e_1$ and $e_2$ are equivalent if
    • Neither $e_1$ nor $e_2$ results in an exception or a deadlock or
    • both $e_1$ and $e_2$ fail for the same reason

• Very liberal, but practical definition
  – It errs on the side of caution to avoid false positives
  – A study of 105 real-world concurrency bugs found that 62% of them lead to a crash or a deadlock
Thread safety oracle

- If a test results in an exception or a deadlock, the oracle iterates over all valid linearizations of the test and checks whether a sequential execution of it causes the exact same failure.

- No such linearization found

  => concurrency bug!
Evaluation

• The authors analyzed classes from six popular libraries, including the Java Standard library and Apache Commons DBCP

• Found 15 bugs in classes marked as thread safe
  – 6 were previously unknown
  – 12 bugs revealed by implicit exceptions
  – Time to find bugs ranged from a few seconds to over 8 hours
```java
StringBuffer sb = new StringBuffer("abc");

sb.insert(1, sb); sb.deleteCharAt(0);
```

Result: `IndexOutOfBoundsException` in Thread 1
Concluding remarks

• The good
  – Full automation of test generation, execution and analysis is a very, very good thing
  – No false positives or duplicate error reports
  – Effective

• The bad
  – Current implementation is not terribly efficient
  – Doesn’t catch “subtle” bugs
  – Humans don’t program uniformly at random

Full source code and on-line version available at www.thread-safe.org
Questions?
Algorithm 1: Returns a concurrent test \((p, s_1, s_2)\)

1. \(\mathcal{P}\): set of prefixes \(\triangleright\) global variables
2. \(\mathcal{M}\): maps a prefix to suffixes
3. \(\mathcal{T}\): set of ready-to-use tests
4. if \(|\mathcal{T}| > 0\) then
5. \(\text{return } \text{randRemove}(\mathcal{T})\)
6. if \(|\mathcal{P}| < \text{maxPrefixes}\) then \(\triangleright\) create a new prefix
7. \(p \leftarrow \text{instantiateCUTTask}(\text{empty call sequence})\)
8. if \(p = \text{failed}\) then
9. \(\text{if } \mathcal{P} = \emptyset \text{ then}\)
10. \(\text{fail(”cannot instantiate CUT”)}\)
11. \(\text{else}\)
12. \(p \leftarrow \text{randTake}(\mathcal{P})\)
13. \(\text{else}\)
14. for \(i \leftarrow 1, \text{maxStateChangerTries}\) do
15. \(p_{\text{ext}} \leftarrow \text{callCUTTask}(p)\)
16. if \(p_{\text{ext}} \neq \text{failed}\) then
17. \(p \leftarrow p_{\text{ext}}\)
18. \(\mathcal{P} \leftarrow \mathcal{P} \cup \{p\}\)
19. \(\text{else}\)
20. \(p \leftarrow \text{randTake}(\mathcal{P})\)
21. \(s_1 \leftarrow \text{empty call sequence}\) \(\triangleright\) create a new suffix
22. for \(i \leftarrow 1, \text{maxCUTCallTries}\) do
23. \(s_{1,\text{ext}} \leftarrow \text{callCUTTask}(s_1, p)\)
24. if \(s_{1,\text{ext}} \neq \text{failed}\) then
25. \(s_1 \leftarrow s_{1,\text{ext}}\)
26. \(\mathcal{M}(p) \leftarrow \mathcal{M}(p) \cup \{s_1\}\)
27. for all \(s_2 \in \mathcal{M}(p)\) do \(\triangleright\) one test for each pair of suffixes
28. \(\mathcal{T} \leftarrow \mathcal{T} \cup \{(p, s_1, s_2)\}\)
29. \(\text{return } \text{randRemove}(\mathcal{T})\)
Algorithm 2 Checks whether a test \((p, s_1, s_2)\) exposes a thread safety bug

1: repeat
2: \(e_{(p,s_1,s_2)} \leftarrow \text{execute}(p, s_1, s_2)\)
3: if failed\((e_{(p,s_1,s_2)})\) then
4: \(\text{seqFailed} \leftarrow \text{false}\)
5: for all \(l \in \mathcal{L}(p, s_1, s_2)\) do
6: \(\text{if seqFailed} = \text{false} \text{ then}\)
7: \(e_l \leftarrow \text{execute}(l)\)
8: \(\text{if failed}(e_l) \land \text{sameFailure}(e_{(p,s_1,s_2)}, e_l) \text{ then}\)
9: \(\text{seqFailed} \leftarrow \text{true}\)
10: \(\text{if seqFailed} = \text{false} \text{ then}\)
11: report bug \(e_{(p,s_1,s_2)}\) and exit
12: until maxConcExecs reached
class StringBuffer {
    StringBuffer(String s) {
        // initialize with the given String
    }
    synchronized void deleteCharAt(int index) {
        // modify while holding the lock
    }
    void insert(int dstOffset, CharSequence s) {
        int l = s.length();
        // BUG: l may change
        this.insert(dstOffset, s, 0, l);
    }
    synchronized void insert(int dstOffset,
                              CharSequence s, int start, int end) {
        // modify while holding the lock
    }
}