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

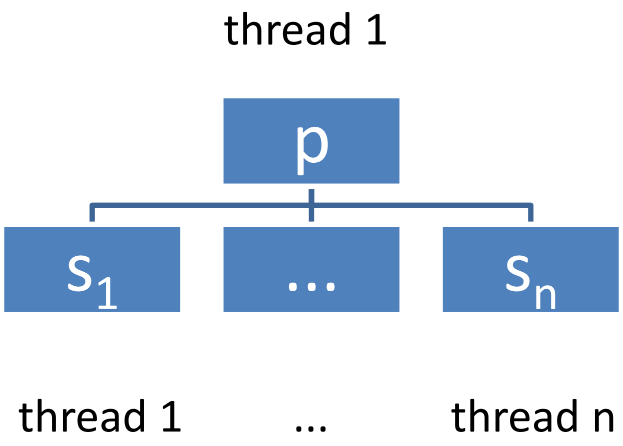
1. Generate concurrent test
2. Execute test repeatedly
3. Check whether thread safety violation caused a test to fail
4. Go back go #1

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

```
StringBuffer sb = new StringBuffer("abc");
```

Thread 1 | Thread 2

```
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)

```
$\mathcal{P}$: set of prefixes                                   ▷ global variables
$\mathcal{M}$: maps a prefix to suffixes
$\mathcal{T}$: set of ready-to-use tests
if $|\mathcal{T}| > 0$ then
    return $randRemove(\mathcal{T})$
if $|\mathcal{P}| < maxPrefixes$ then                             ▷ create a new prefix
    $p \leftarrow instantiateCUTTask$(empty call sequence)
    if $p = failed$ then
        if $\mathcal{P} = \emptyset$ then
            $fail$("cannot instantiate CUT")
        else
            $p \leftarrow randTake(\mathcal{P})$
    else
        for $i \leftarrow 1, maxStateChangerTries$ do
            $p_{ext} \leftarrow callCUTTask(p)$
            if $p_{ext} \neq failed$ then
                $p \leftarrow p_{ext}$
    $\mathcal{P} \leftarrow \mathcal{P} \cup \{p\}$
else
    $p \leftarrow randTake(\mathcal{P})$
$s_1 \leftarrow$ empty call sequence                              ▷ create a new suffix
for $i \leftarrow 1, maxCUTCallTries$ do
    $s_{1,ext} \leftarrow callCUTTask(s_1, p)$
    if $s_{1,ext} \neq failed$ then
        $s_1 \leftarrow s_{1,ext}$
$\mathcal{M}(p) \leftarrow \mathcal{M}(p) \cup \{s_1\}$
for all $s_2 \in \mathcal{M}(p)$ do                               ▷ one test for each pair of suffixes
    $\mathcal{T} \leftarrow \mathcal{T} \cup \{(p, s_1, s_2)\}$
return $randRemove(\mathcal{T})$
```

Algorithm 2 Checks whether a test (p, s_1, s_2) exposes a thread safety bug

repeat
  $e_{(p,s_1,s_2)} \leftarrow execute(p, s_1, s_2)$
  if $failed(e_{(p,s_1,s_2)})$ **then**
    $seqFailed \leftarrow$ false
    for all $l \in \mathcal{L}(p, s_1, s_2)$ **do**
      if $seqFailed =$ false **then**
        $e_l \leftarrow execute(l)$
        if $failed(e_l) \wedge sameFailure(e_{(p,s_1,s_2)}, e_l)$ **then**
          $seqFailed \leftarrow$ true
    if $seqFailed =$ false **then**
      report bug $e_{(p,s_1,s_2)}$ and exit
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
  }
}
```