FULLY AUTOMATIC AND PRECISE DETECTION OF THREAD SAFETY VIOLATIONS

PLDI 2012
by
Michael Pradel and Thomas R. Gross
ETH Zurich

presented by
Martin Aigner
University of Salzburg
May 2, 2013
OVERVIEW

- The problem of testing multi-threaded code
- Proposed solution: automatically generate test cases
- Evaluation of the results
Testing for thread-safety

• Consider a (wanna-be thread-safe) class in an OO language...

• Try to argue about it’s thread-safety... How do you do that?
  • Verify correct concurrent behavior?
    • Too expensive in most cases

• Write tests? How? Where to start? Where to end?
  • Also expensive
class StringBuffer implements CharSequence {

    /* init instance with given string */
    StringBuffer(String s) {...}

    /* modify this holding a lock */
    synchronized void deleteCharAt (int index) {...}

    /* modify this while holding a lock */
    synchronized void insert (int dstOffset, CharSequence s, int start, int end) {...}

    /* convenience overloading */
    void insert (int dstOffset, CharSequence s) {
        this.insert(dstOffset, s, 0, s.length());
    }
}
class StringBuffer implements CharSequence {

    /* init instance with given string */
    StringBuffer(String s) {...}

    /* modify this holding a lock */
    synchronized void deleteCharAt (int index) {...}

    /* modify this while holding a lock */
    synchronized void insert (int dstOffset, CharSequence s, int start, int end) {...}

    /* convenience overloading */
    void insert (int dstOffset, CharSequence s) {
        this.insert(dstOffset, s, 0, s.length());
    }
}

main: StringBuffer sb1 = new StringBuffer(“thread-safe”);
main: StringBuffer sb2 = new StringBuffer(“not”);

Thread 1
sb1.insert(0, sb2);

Thread 2
sb1.deleteCharAt(0);

Result?
class StringBuffer implements CharSequence {

    /* init instance with given string */
    StringBuffer(String s) {...}

    /* modify this holding a lock */
    synchronized void deleteCharAt (int index) {...}

    /* modify this while holding a lock */
    synchronized void insert (int dstOffset, CharSequence s, int start, int end) {...}

    /* convenience overloading */
    void insert (int dstOffset, CharSequence s) {
        this.insert(dstOffset, s, 0, s.length());
    }
}

main: StringBuffer sb1 = new StringBuffer("thread-safe");

        Thread 1
        sb1.insert(0, sb1);

        Thread 2
        sb1.deleteCharAt(0);

Result?
Problems with testing

• In general:
  • Cover the configuration space
  • Observe success of failure

• For concurrent tests:
  • Find out if a concurrent execution meets a sequential specification
PROPOSED SOLUTION

Create, execute, and validate test cases automatically
REQUIREMENTS

• We need:
  • The class under test (CUT)
  • A set of auxiliary classes the CUT depends on
  • A test that maybe triggers a bug (thread-safety violation)
  • An execution environment that exposes the bug
  • An oracle that recognizes an erroneous execution
Generating concurrent tests

- Definitions: call sequence, prefix, suffixes
  
  - A call sequence is a n-tuple of calls \((c_1, ..., c_n)\) where a call is a method together with input and output parameters
    
    - Prefix: a call sequence that is executed sequentially to “grow” an object
      
      - Suffixes: a set of call sequences that are executed concurrently on an object to trigger a concurrency bug
Generating Concurrent Tests

- Definitions: test

- A test is a triple \((p, s_1, s_2)\) where \(p\) is a prefix and \(s_1\) and \(s_2\) are suffixes

```
  p_1
     /\  \\
  /   \  \\
S_{11}  S_{21}
     /\  \\
  /   \  \\
S_{1j}  S_{2k}
```
**Example: Concurrent Test**

```java
main: StringBuffer sb1 = new StringBuffer("thread-safe");

Thread 1
sb1.insert(0, sb1);

Thread 2
sb1.deleteCharAt(0);
```

Prefix: `p`

Suffixes:
- `s1`
- `s2`
Tasks: Creating Call Sequences

• A task takes a call sequence $s_{in} = (c_1, ..., c_i)$ and returns a call sequence $s_{out} = (c_1, ..., c_i, c_j, ..., c_n)$

• 3 types of tasks:
  
  • instantiateCUTTask: appends a call to instantiate the CUT
  
  • callCUTTask: appends a call to the CUT instance
  
  • parameterTask: provides a typed parameter by either:
    
    • selecting a previous output parameter
    
    • appending a call that returns the parameter
Test generation algorithm

• Three global variables:
  • $P$: a set of prefixes
  • $M$: a map assigning a prefix to its set of suffixes
  • $T$: a set of already generated tests
Test generation algorithm

• Step 1: create a new prefix
  • invokes `instantiateCUTTask` to create an instance of CUT
  • repeatedly invokes `callCUTTask` to extend the prefix
  • the `parameterTask` is invoked whenever a call requires a parameter
Test generation algorithm

• Step 2: create a suffix for a prefix
  • repeatedly invokes `callCUTTask` and use the output parameters of the prefix as input the suffix
  • again, the `parameterTask` is invoked whenever a call requires a parameter
  • add the suffix to M for the given prefix
Test generation algorithm

• Step 3: test creation

• create tests based on the prefix, the suffix and all its other suffixes (obtained from M) and store them in T

• randomly return a test from T
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: \hspace{-2em} \textbf{return} randRemove(\(\mathcal{T}\))
6: if \(|\mathcal{P}| < \text{maxPrefixes}\) then \(\triangleright\) create a new prefix
7: \hspace{-2em} \(p \leftarrow \text{instantiateCUTTask}(\text{empty call sequence})\)
8: if \(p = \text{failed}\) then
9: if \(\mathcal{P} = \emptyset\) then
10: \hspace{-2em} \text{fail}(“cannot instantiate \text{CUT}”)
11: else
12: \hspace{-2em} \(p \leftarrow \text{randTake}(\mathcal{P})\)
13: else
14: \hspace{-2em} for \(i \leftarrow 1, \text{maxStateChangerTries}\) do
15: \hspace{2em} \(p_{\text{ext}} \leftarrow \text{callCUTTask}(p)\)
16: \hspace{2em} if \(p_{\text{ext}} \neq \text{failed}\) then
17: \hspace{4em} \(p \leftarrow p_{\text{ext}}\)
18: \hspace{2em} \(\mathcal{P} \leftarrow \mathcal{P} \cup \{p\}\)
19: else
20: \hspace{2em} \(p \leftarrow \text{randTake}(\mathcal{P})\)
21: \(s_1 \leftarrow \text{empty call sequence}\) \(\triangleright\) create a new suffix
22: for \(i \leftarrow 1, \text{maxCUTCallTries}\) do
23: \hspace{2em} \(s_{1,\text{ext}} \leftarrow \text{callCUTTask}(s_1, p)\)
24: \hspace{2em} if \(s_{1,\text{ext}} \neq \text{failed}\) then
25: \hspace{4em} \(s_1 \leftarrow s_{1,\text{ext}}\)
26: \hspace{2em} \(\mathcal{M}(p) \leftarrow \mathcal{M}(p) \cup \{s_1\}\)
27: for all \(s_2 \in \mathcal{M}(p)\) do \(\triangleright\) one test for each pair of suffixes
28: \hspace{2em} \(\mathcal{T} \leftarrow \mathcal{T} \cup \{(p, s_1, s_2)\}\)
29: \textbf{return} randRemove(\(\mathcal{T}\))

Step 1

Step 2

Step 3

the parameters \text{maxPrefixes}, \text{maxStateChangerTries} and \text{maxCUTCallTries} are heuristically defined limits.
A class is said to be thread-safe (in Java methodology) if:

- multiple threads can use it without synchronization

- and the observed behavior of a concurrent execution is equivalent to and sequential execution of a linearization of the calls preserving per-thread ordering
**EXAMPLE**

```java
main: CopyOnWriteArrayList l = new CopyOnWriteArrayList();

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>l.add(&quot;a&quot;);</td>
<td>l.add(&quot;b&quot;);</td>
</tr>
<tr>
<td>println(l.toString());</td>
<td></td>
</tr>
</tbody>
</table>

Results: [a], [a,b], or [b,a]

Linearizations:

- add("a"); → println(); → add("b"); gives [a]
- add("a"); → add("b"); → println(); gives [a, b]
- add("b"); → add("a"); → println(); gives [b, a]
Definitions

• operator $\oplus$ concatenates call sequences

• For calls $c$ and $c'$, the notion $c \rightarrow c'$ indicates that call $c$ precedes $c'$
Linearization

**Definition 1 (Linearization).** For a test \((p, s_1, s_2)\), let \(\mathcal{P}_{12}\) be the set of all permutations of the call sequence \(s_1 \oplus s_2\). The set of linearizations of the test is:

\[
\mathcal{L}_{(p, s_1, s_2)} = \{ p \oplus s_{12} \mid s_{12} \in \mathcal{P}_{12} \land \]

\[
(\forall c, c' \ (c \rightarrow_{s_1} c' \Rightarrow c \rightarrow_{s_{12}} c') \land \]

\[
(c \rightarrow_{s_2} c' \Rightarrow c \rightarrow_{s_{12}} c'))\}
\]

Intuitively: a linearization of a test \((p, s_1, s_2)\) appends to \(p\) all calls from \(s_1\) and from \(s_2\) in a way that preserves the order of calls in \(s_1\) and \(s_2\).
**Definition 2 (Execution).** For a test \((p, s_1, s_2)\), we denote the set of all distinguishable executions of this test as \(\mathcal{E}_{(p,s_1,s_2)}\). Each \(e_{(p,s_1,s_2)} \in \mathcal{E}_{(p,s_1,s_2)}\) represents the sequential execution of \(p\) followed by a concurrent execution of \(s_1\) and \(s_2\). Likewise, we denote the sequential execution of a call sequence \(s\) as \(e_s\).

Note: due to the non-determinism of concurrent executions a single test can have multiple distinguishable executions.
Thread safety

Notation: for executions $e_1$ and $e_2$, the notion $e_1 \equiv e_2$ indicates that $e_1$ and $e_2$ are equivalent (discussed later).

Definition 3 (Thread safety). Let $\mathcal{T}_C$ be the set of all possible tests for a class $C$. $C$ is thread-safe if and only if:

$$\forall (p, s_1, s_2) \in \mathcal{T}_C \ \forall e_{(p,s_1,s_2)} \in \mathcal{E}_{(p,s_1,s_2)}$$

$$\exists l \in \mathcal{L}_{(p,s_1,s_2)} \text{ so that } e_{(p,s_1,s_2)} \equiv e_l$$
Thread safety

Notation: for executions $e_1$ and $e_2$, the notion $e_1 \equiv e_2$ indicates that $e_1$ and $e_2$ are equivalent (discussed later)

**Definition 3 (Thread safety).** Let $T_C$ be the set of all possible tests for a class $C$. $C$ is thread-safe if and only if:

$$\forall (p, s_1, s_2) \in T_C \ \forall e_{(p,s_1,s_2)} \in E_{(p,s_1,s_2)}$$

$$\exists l \in L_{(p,s_1,s_2)} \text{ so that } e_{(p,s_1,s_2)} \equiv e_l$$

**Problem:** this is expensive! All executions times all tests times all linearizations... Let’s try the opposite
Thread-unsafety

We call a class thread-unsafe if:

$$\exists (p, s_1, s_2) \in \mathcal{T}_C \ \exists e_{(p, s_1, s_2)} \in \mathcal{E}_{(p, s_1, s_2)}$$

so that $$\forall l \in \mathcal{L}_{(p, s_1, s_2)} \ e_{(p, s_1, s_2)} \not\equiv e_l$$

Intuitively: the oracle tries to find a test that exposes behavior not possible with any linearization of the test.

Still expensive, but we are lucky this time. We only need to check buggy concurrent executions for equivalence to their linearizations.
**Definition 4 (Equivalence of executions).** 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 (that is, the same type of exception is thrown or both executions end with a deadlock).
Algorithm 2 Checks whether a test \((p, s_1, s_2)\) exposes a thread safety bug

1: repeat
2: \(e_{(p, s_1, s_2)} \leftarrow execute(p, s_1, s_2)\)
3: if failed\(e_{(p, s_1, s_2)}\) then
4: \(seqFailed \leftarrow false\)
5: for all \(l \in L(p, s_1, s_2)\) do
6: if \(seqFailed = false\) then
7: \(e_l \leftarrow execute(l)\)
8: if failed\(e_l\) \&\& sameFailure\(e_{(p, s_1, s_2)}, e_l\) then
9: \(seqFailed \leftarrow true\)
10: if \(seqFailed = false\) then
11: report bug \(e_{(p, s_1, s_2)}\) and exit
12: until maxConcExecs reached

consider only failed concurrent executions

no concurrency bug if linearization fails for the same reason

concurrency bug only if the linearizations did not fail
LIMITATIONS

• Only exceptions and deadlocks are considered

• Testing is still incomplete

• No semantical equivalence of concurrent and linearized executions

• The proposed approach is based on two assumptions:
  • Uncaught exceptions and deadlocks that occur in concurrent executions but not in sequential ones are a problem
  • Sequential execution is deterministic
Contributions

• Fully automatic

• Only true positives are reported

• No (or very little) human effort for testing and evaluation
  • This makes it cheap!

• It does indeed find bugs
EVALUATION

- Tests were performed on six popular code bases
  - Java standard library
  - Apache Commons DBCP
  - XStream (serialization library)
  - LingPipe (text processing toolkit)
  - JFreeChart (chart library)
  - Joda-Time (library for handling data and time)
<table>
<thead>
<tr>
<th>ID</th>
<th>Code base</th>
<th>Class</th>
<th>Declared thread-safe</th>
<th>Found unsafe</th>
<th>Reason for failing</th>
<th>Implicit</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>JDK 1.6.0.27 and 1.7.0</td>
<td>StringBuffer</td>
<td>yes</td>
<td>yes</td>
<td>IndexOutOfBoundsException</td>
<td>yes</td>
</tr>
<tr>
<td>2</td>
<td>JDK 1.6.0.27 and 1.7.0</td>
<td>ConcurrentHashMap</td>
<td>yes</td>
<td>yes</td>
<td>StackOverflowError</td>
<td>yes</td>
</tr>
<tr>
<td>3</td>
<td>Commons DBCP 1.4</td>
<td>SharedPoolDataSource</td>
<td>yes</td>
<td>yes</td>
<td>ConcurrentModificationException</td>
<td>yes</td>
</tr>
<tr>
<td>4</td>
<td>Commons DBCP 1.4</td>
<td>PerUserPoolDataSource</td>
<td>yes</td>
<td>yes</td>
<td>ConcurrentModificationException</td>
<td>yes</td>
</tr>
<tr>
<td>5</td>
<td>XStream 1.4.1</td>
<td>XStream</td>
<td>yes</td>
<td>yes</td>
<td>NullPointerException</td>
<td>no</td>
</tr>
<tr>
<td>6</td>
<td>LingPipe 4.1.0</td>
<td>MedlineSentenceModel</td>
<td>yes</td>
<td>yes</td>
<td>IllegalStateException</td>
<td>no</td>
</tr>
<tr>
<td>7</td>
<td>JDK 1.1</td>
<td>BufferedInputStream</td>
<td>yes</td>
<td>yes</td>
<td>NullPointerException</td>
<td>yes</td>
</tr>
<tr>
<td>8</td>
<td>JDK 1.4.1</td>
<td>Logger</td>
<td>yes</td>
<td>yes</td>
<td>NullPointerException</td>
<td>yes</td>
</tr>
<tr>
<td>9</td>
<td>JDK 1.4.2</td>
<td>SynchronizedMap</td>
<td>yes</td>
<td>yes</td>
<td>Deadlock</td>
<td>yes</td>
</tr>
<tr>
<td>10</td>
<td>JFreeChart 0.9.8</td>
<td>TimeSeries</td>
<td>yes</td>
<td>yes</td>
<td>NullPointerException</td>
<td>yes</td>
</tr>
<tr>
<td>11</td>
<td>JFreeChart 0.9.8</td>
<td>XYSeries</td>
<td>yes</td>
<td>yes</td>
<td>ConcurrentModificationException</td>
<td>yes</td>
</tr>
<tr>
<td>12</td>
<td>JFreeChart 0.9.12</td>
<td>NumberAxis</td>
<td>yes</td>
<td>yes</td>
<td>IllegalArgumentException</td>
<td>no</td>
</tr>
<tr>
<td>13</td>
<td>JFreeChart 1.0.1</td>
<td>PeriodAxis</td>
<td>yes</td>
<td>yes</td>
<td>IllegalArgumentException</td>
<td>no</td>
</tr>
<tr>
<td>14</td>
<td>JFreeChart 1.0.9</td>
<td>XYPlot</td>
<td>yes</td>
<td>yes</td>
<td>ConcurrentModificationException</td>
<td>yes</td>
</tr>
<tr>
<td>15</td>
<td>JFreeChart 1.0.13</td>
<td>Day</td>
<td>yes</td>
<td>yes</td>
<td>NumberFormatException</td>
<td>yes</td>
</tr>
</tbody>
</table>

### Previously unknown bugs:

- **(1)** JDK 1.6.0.27 and 1.7.0
- **(2)** JDK 1.6.0.27 and 1.7.0
- **(3)** Commons DBCP 1.4
- **(4)** Commons DBCP 1.4
- **(5)** XStream 1.4.1
- **(6)** LingPipe 4.1.0

### Known bugs:

- **(7)** JDK 1.1
- **(8)** JDK 1.4.1
- **(9)** JDK 1.4.2
- **(10)** JFreeChart 0.9.8
- **(11)** JFreeChart 0.9.8
- **(12)** JFreeChart 0.9.12
- **(13)** JFreeChart 1.0.1
- **(14)** JFreeChart 1.0.9
- **(15)** JFreeChart 1.0.13

### Automatic classification of classes as thread-unsafe:

- **(16)** Joda-Time 2.0
- **(17)** Joda-Time 2.0
- **(18)** Joda-Time 2.0
- **(19)** Joda-Time 2.0
- **(20)** Joda-Time 2.0
- **(21)** Joda-Time 2.0
- **(22)** Joda-Time 2.0

<table>
<thead>
<tr>
<th>Code base</th>
<th>Class</th>
<th>Declared thread-safe</th>
<th>Found unsafe</th>
<th>Reason for failing</th>
<th>Implicit</th>
</tr>
</thead>
<tbody>
<tr>
<td>Joda-Time 2.0</td>
<td>DateTimeFormatterBuilder</td>
<td>no</td>
<td>yes</td>
<td>IndexOutOfBoundsException (10x)</td>
<td>yes</td>
</tr>
<tr>
<td>Joda-Time 2.0</td>
<td>DateTimeParserBucket</td>
<td>no</td>
<td>yes</td>
<td>IllegalArgumentException (9x)</td>
<td>no</td>
</tr>
<tr>
<td>Joda-Time 2.0</td>
<td>DateTimeTimeZoneBuilder</td>
<td>no</td>
<td>yes</td>
<td>NullPointerException (6x)</td>
<td>yes</td>
</tr>
<tr>
<td>Joda-Time 2.0</td>
<td>DateTimeZoneBuilder</td>
<td>no</td>
<td>yes</td>
<td>ArrayIndexOutOfBoundsException (2x)</td>
<td>yes</td>
</tr>
<tr>
<td>Joda-Time 2.0</td>
<td>DateTimeZoneBuilder</td>
<td>no</td>
<td>yes</td>
<td>IllegalFieldValueException (2x)</td>
<td>yes</td>
</tr>
<tr>
<td>Joda-Time 2.0</td>
<td>MutableDateTime</td>
<td>no</td>
<td>yes</td>
<td>IllegalFieldValueException (9x)</td>
<td>no</td>
</tr>
<tr>
<td>Joda-Time 2.0</td>
<td>MutableDateTime</td>
<td>no</td>
<td>yes</td>
<td>ArithmeticException (1x)</td>
<td>yes</td>
</tr>
<tr>
<td>Joda-Time 2.0</td>
<td>MutableInterval</td>
<td>no</td>
<td>yes</td>
<td>IllegalArgumentException (10x)</td>
<td>no</td>
</tr>
<tr>
<td>Joda-Time 2.0</td>
<td>MutablePeriod</td>
<td>no</td>
<td>yes</td>
<td>ArithmeticException (10x)</td>
<td>yes</td>
</tr>
<tr>
<td>Joda-Time 2.0</td>
<td>PeriodFormatterBuilder</td>
<td>no</td>
<td>yes</td>
<td>ConcurrentModificationException (5x)</td>
<td>yes</td>
</tr>
<tr>
<td>Joda-Time 2.0</td>
<td>PeriodFormatterBuilder</td>
<td>no</td>
<td>yes</td>
<td>IndexOutOfBoundsException (4x)</td>
<td>yes</td>
</tr>
<tr>
<td>Joda-Time 2.0</td>
<td>ZoneInfoCompiler</td>
<td>no</td>
<td>no</td>
<td>(stopped after 24h)</td>
<td>–</td>
</tr>
</tbody>
</table>
The sequence is not extended.

sequence exceeds the available memory, an exception is thrown and maintain significant amounts of state. If executing a generated call test generator selects methods randomly and therefore does not effort. The approach has very moderate memory requirements. The running time of our implementation with additional engineering effort. This time to be still acceptable. Section 5 outlines ways to reduce this time to be still acceptable.

Several years in one of the most popular Java libraries, we consider bug 8 (JDK's

require several hours of computation time, with up to 8.2 hours for problems, the analysis takes only a few seconds. Other classes runs. Most of the problems are found within one hour. For several average, and maximum time taken to find the problem over ten required to find the problem. The vertical axis gives the minimum, shows the IDs from Table 1, sorting the classes by the average time takes to trigger the problems from Table 1. The horizontal axis shows that both properties are important [3].

Experience from applying automated bug finding techniques in industry

as only input and produces true positives as only output. Expe-

analysis requires the source code or byte code of the classes under

Using our approach involves minimal human effort, because the

6.4 Effort of Using the Analysis

class. For the 33 classes that are documented as thread-safe, no

files, it cannot detect problems caused by concurrent usages of the

Since the thread-safety oracle does not check the integrity of such

8.1 Finding Concurrency Bugs

Properties of bug finding techniques:

- whether static or dynamic analysis is used
- the on 4 criteria: whether static or dynamic analysis is used, the

Table 2 compares this work with other bug finding techniques based

8. Related Work

obviously wrong results.

concurrent calls to the same object. Third, the approach is limited

port for multi-object bugs. Although the generated tests combine

legal, concurrent behavior. Second, the approach has limited sup-

tice, but in principle a class could throw exceptions as part of its

desirable. We found this “implicit specification” to be true in prac-

First, the approach assumes that exceptions and deadlocks are un-

scheduling may reduce the number of tests for hitting a bug. Combining our work with techniques to control

suggests that the task of executing a bug-exposing test with the

lion for bug 4. A manual inspection of bugs requiring many tests

expose the problem. Other bugs require more tests, up to 17 mil-

ber of tests (several hundreds or even less than hundred) suffices to

analysis generates and executes before hitting a bug. Figure 6b shows

A question related to running time is how many tests the anal-

Figure 6: Effort required to trigger a thread-safety problem.

Time (min/avg/max minutes)

CUTs (sorted by avg. time)

(a) Supposedly thread-safe class.

(b) Tests generated before triggering a thread-safety problem.

Execution of a generated concurrent test exposing a thread-safety bug.

Figure 5: Concurrency bug in Apache Commons DBCP.
The sequence is not extended.

If executing a generated call test generator selects methods randomly and therefore does not maintain significant amounts of state. If executing a generated call, the approach has very moderate memory requirements. The running time of our implementation with additional engineering effort. The approach has very moderate memory requirements. The computational effort of our implementation is acceptable this time to be still acceptable. Section 5 outlines ways to reduce this time to be still acceptable.

For the 33 classes that are documented as thread-safe, no problems are found after running the analysis for 24 hours. Since the thread-safety oracle does not check the integrity of such files, it cannot detect problems caused by concurrent usages of the instanceMap. Given that the bug remained unnoticed for several years in one of the most popular Java libraries, we consider bug 8 (JDK's InstanceKeyObjectFactory) to be obviously wrong results. Given that the bug remained unnoticed for several years in one of the most popular Java libraries, we consider bug 8 (JDK's InstanceKeyObjectFactory) to be obviously wrong results.

Figure 6 shows the IDs from Table 1, sorting the classes by the average time required to find the problem. The vertical axis gives the minimum, average, and maximum time taken to find the problem over ten runs. The horizontal axis shows that both properties are important [3].

Figure 6b shows how long the analysis requires the source code or byte code of the classes under analysis. Analysis requires the source code or byte code of the classes under analysis. Analysis requires the source code or byte code of the classes under analysis. Analysis requires the source code or byte code of the classes under analysis. Analysis requires the source code or byte code of the classes under analysis.

ConcurrentModificationException in Thread 1. RegisterNewInstance(this);

Thread 2

DataSourceName("a");

ConcurrentModificationException in Thread 1

Execution of a generated concurrent test exposing a thread-safety bug.
detection of thread safety violations. In Proceedings of the 33rd ACM SIGPLAN
conference on Programming Language Design and Implementation (PLDI '12).
ACM, New York, NY, USA, 521-530.
THANKS A LOT