Efficient Detection of Thread Safety Violations via Coverage-guided Generation of Concurrent Unit Tests

Ankit Choudhary

Shan Lu

Michael Pradel

TECHNISCHE UNIVERSITÄT DARMSTADT

SOLA SoftwareLab

THE UNIVERSITY OF CHICAGO

Thread Safety

"A class is thread-safe if it behaves correctly when accessed from multiple threads, regardless of the scheduling or interleaving of the execution of those threads by the runtime environment, and with no additional synchronization or coordination on the part of the calling code."

- Java Concurrency in Practice

Thread Safety Bug - Example

```
public class IntegerList {
        protected int array[] = ...;
        protected int index = 0;
        public synchronized void add(int num) {
                if(array != null) {
                        if(index == array.length) {
                                resize();
                        }
                }
        }
        public void close() {
                array = null;
        }
}
```

Thread Safety Bug - Example

```
public class IntegerList {
        protected int array[] = ...;
        protected int index = 0;
        public synchronized void add(int num) {
                if(array != null) {
                        if(index == array.length) {
                                resize();
                        }
                }
        }
        public void close() {
                array = null;
        }
}
```

🟩 Thread 1

🟦 Thread 2

Thread Safety Bug - Example

```
public class IntegerList {
    protected int array[] = ..
    protected int index = 0;
    public synchronized void           num) {
        if(array != null) {
            if(index == array.length) {
                resize();
            }
        }
    }
    public void close() {
        array = null;
    }
}
```

true

Thread 1

Thread 2

Thread Safety Bug - Example

```
public class IntegerList {
        protected int array[] = ..
        protected int index = 0;
        public synchronized void              num) {
                if(array != null) {
                        if(index == array.length) {
                                resize();
                        }
                }
        }
        public void close() {
                array = null;
        }
}
```

null

true

Thread 1

Thread 2

Thread Safety Bug - Example

```
public class IntegerList {
        protected int array[] = ...;
        protected int index = 0;
        public synchronized void add(int num) {
                if(array != null) {
                        if(index == array.length) {
                                resize();
                        }
                }
        }
        public void close() {
                array = null;
        }
}
```

null

NullPointerException

Thread 1

Thread 2

Finding Concurrency Bugs

Generation of Multithreaded Tests

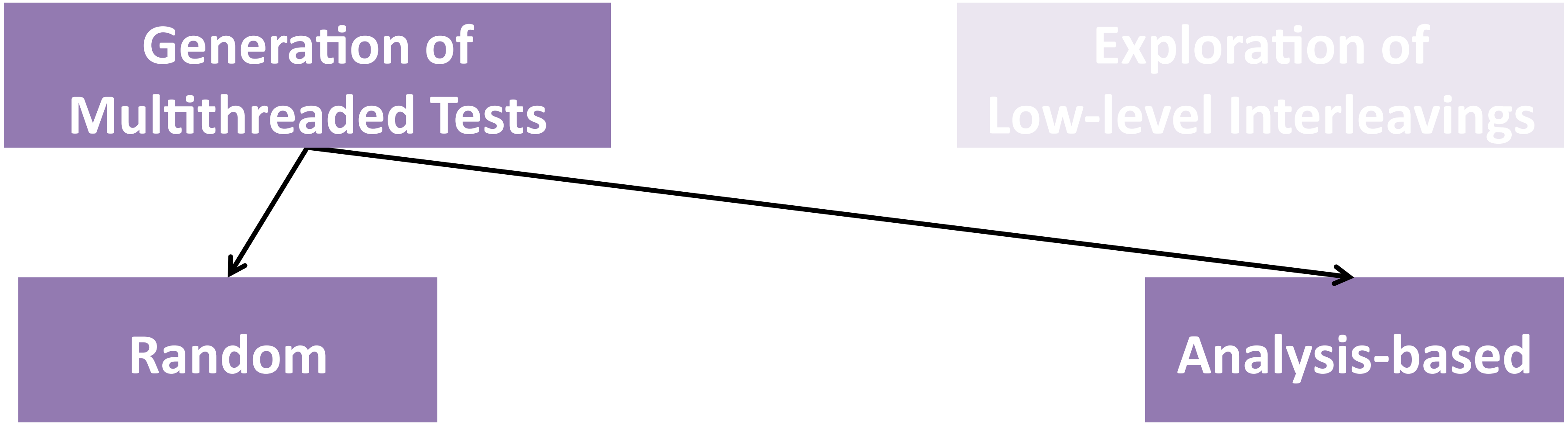
Exploration of Low-level Interleavings

Finding Concurrency Bugs

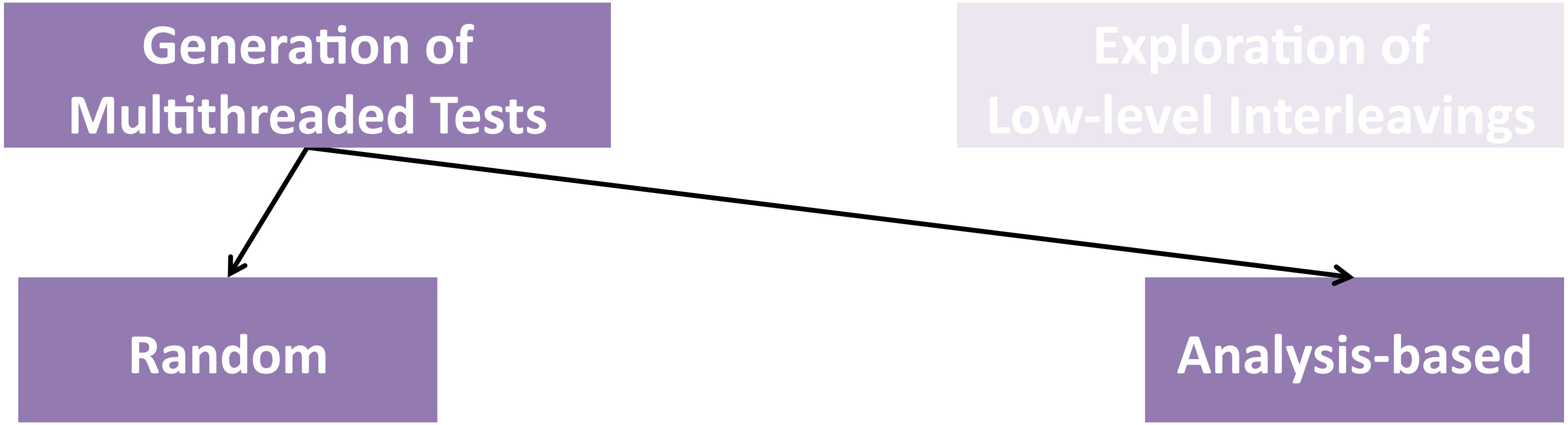
Generation of Multithreaded Tests

Exploration of Low-level Interleavings

Finding Concurrency Bugs


Generation of Multithreaded Tests
Exploration of Low-level Interleavings
Random
Analysis-based

Finding Concurrency Bugs



Generation of Multithreaded Tests

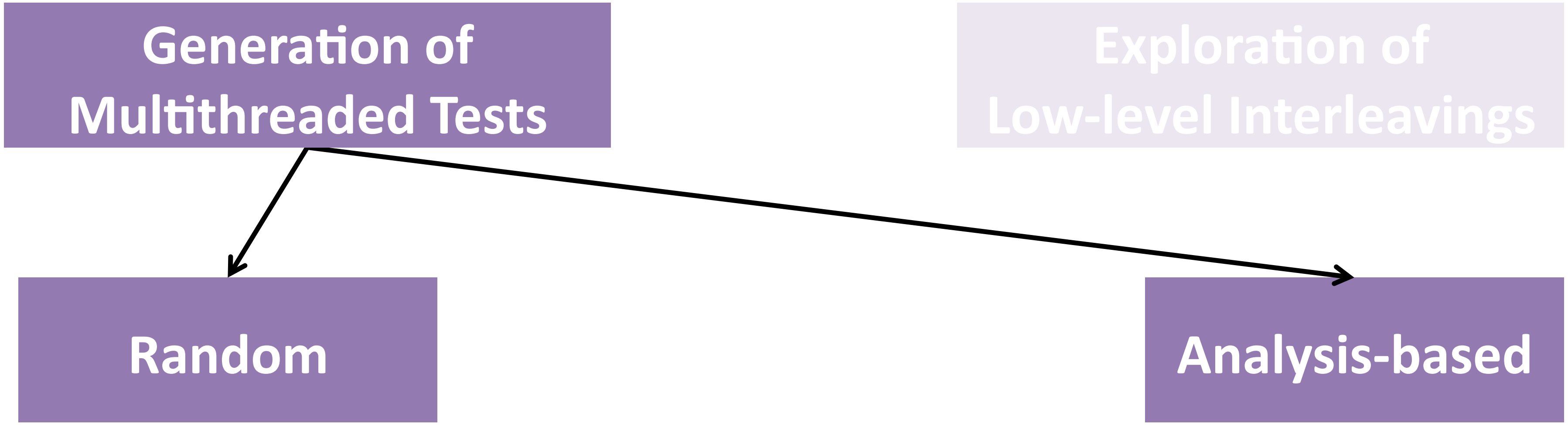
Exploration of Low-level Interleavings

Random

Analysis-based

- + Simple and inexpensive.
- - Repeatedly generates same test.
- - Does not consider locks / synchronization (use static analysis).

Finding Concurrency Bugs



Generation of Multithreaded Tests

Exploration of Low-level Interleavings

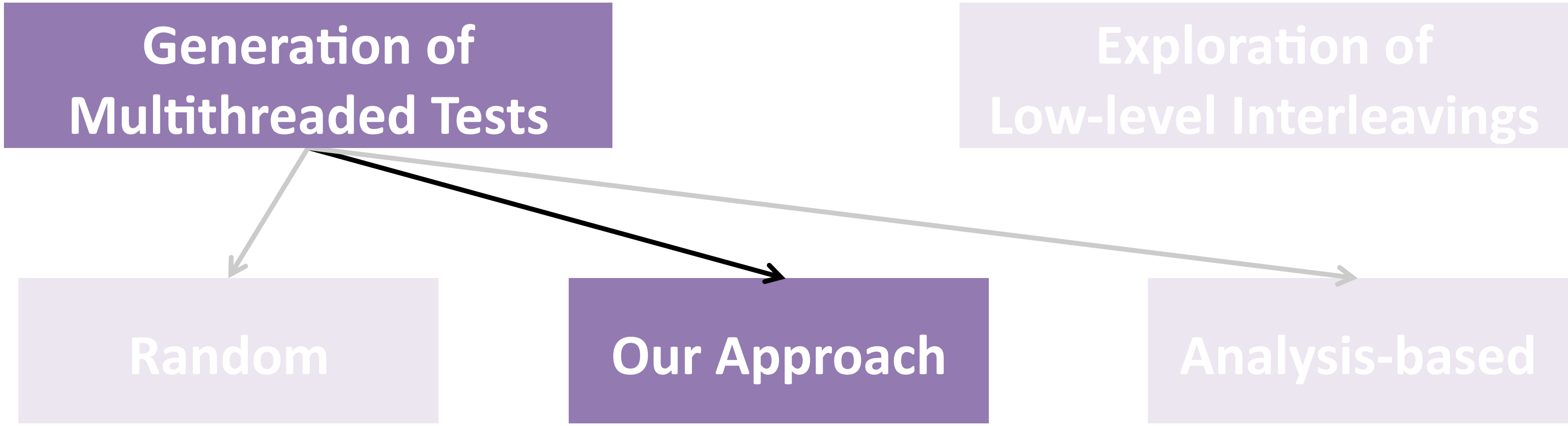
Random

Analysis-based

+ Simple and inexpensive.
- Repeatedly generates same test.
- Does not consider locks / synchronization (use static analysis).

+ Tests directed towards finding bug.
- Costly (time).
- Focuses on a particular type of bug (race conditions, atomicity violations, or deadlocks).

This Talk



Generation of Multithreaded Tests

Exploration of Low-level Interleavings

Random

Our Approach

Analysis-based

+ Simple and inexpensive.

~~- Repeatedly generates same test.~~

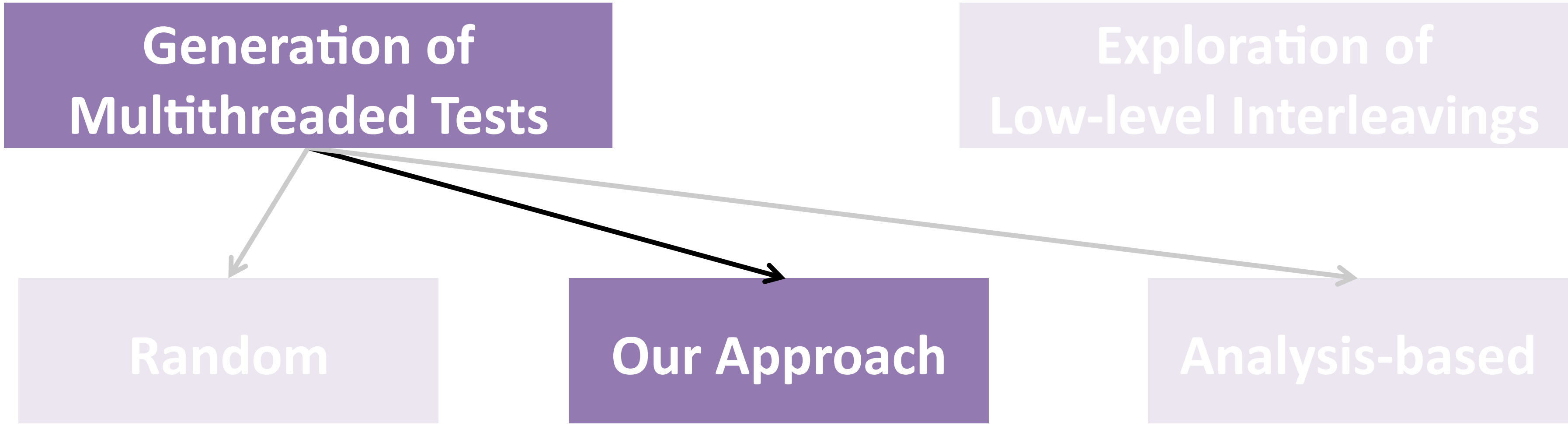
~~- Does not consider locks / synchronization (use static analysis)~~. Dynamically assigns lower priority to methods with locks / synchronization.

+ Tests directed towards ~~finding bug~~ not yet generated ones.

~~- Costly (time)~~.

+ Focuses on ~~a particular type~~ all types of bug ~~(race conditions, atomicity violations, or deadlocks)~~ that can lead to exception or deadlock.

This Talk



Generation of Multithreaded Tests

Exploration of Low-level Interleavings

Random

Our Approach

Analysis-based

- + Simple and inexpensive.
- – ~~Repeatedly generates same test.~~
- – ~~Does~~ ~~sync~~ ~~analysis~~). Dynamically assigns lower priority to methods with locks / synchronization.

- + Tests directed towards ~~finding bug~~ not yet generated ones.
- ~~typ~~e of bug ~~(race conditions, atomicity violations, or deadlocks)~~ that can lead to exception or deadlock.

Best of Both Worlds!

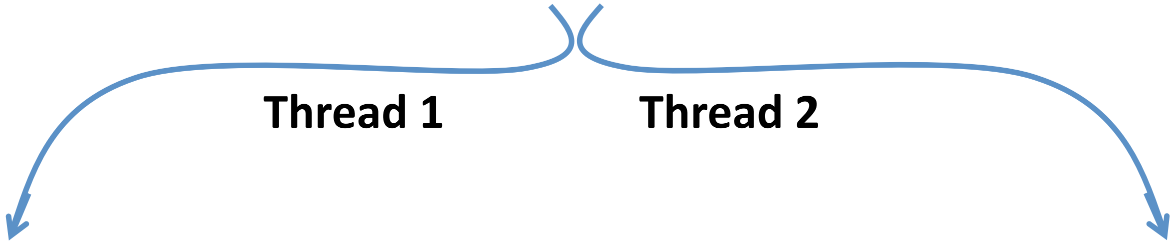
A Concurrent Test

A Concurrent Test

```
IntegerList il = new IntegerList();
```

A Concurrent Test

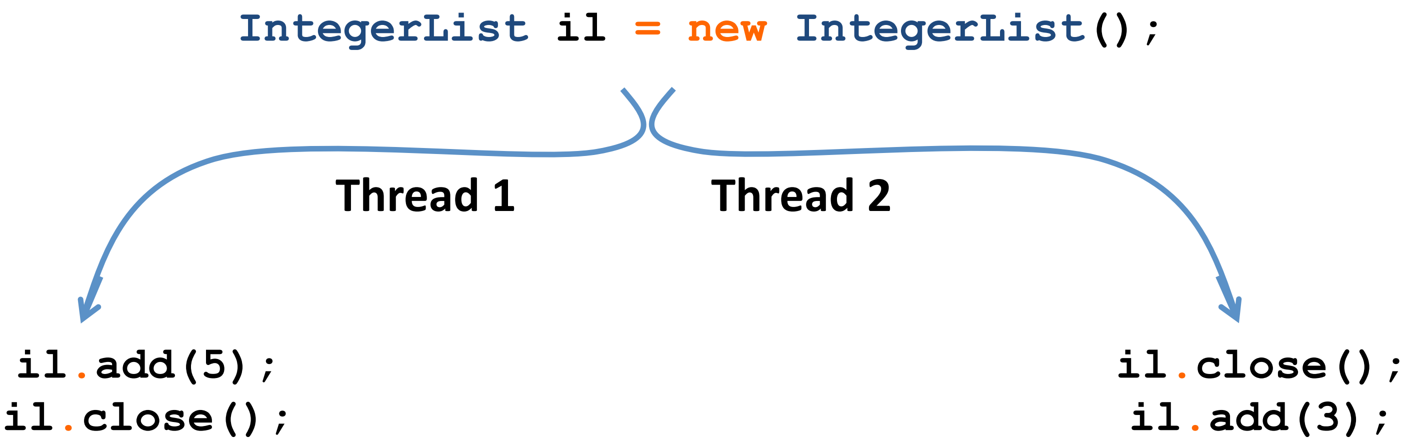
```
IntegerList il = new IntegerList();
```



Thread 1

Thread 2

A Concurrent Test



```
IntegerList il = new IntegerList();
```

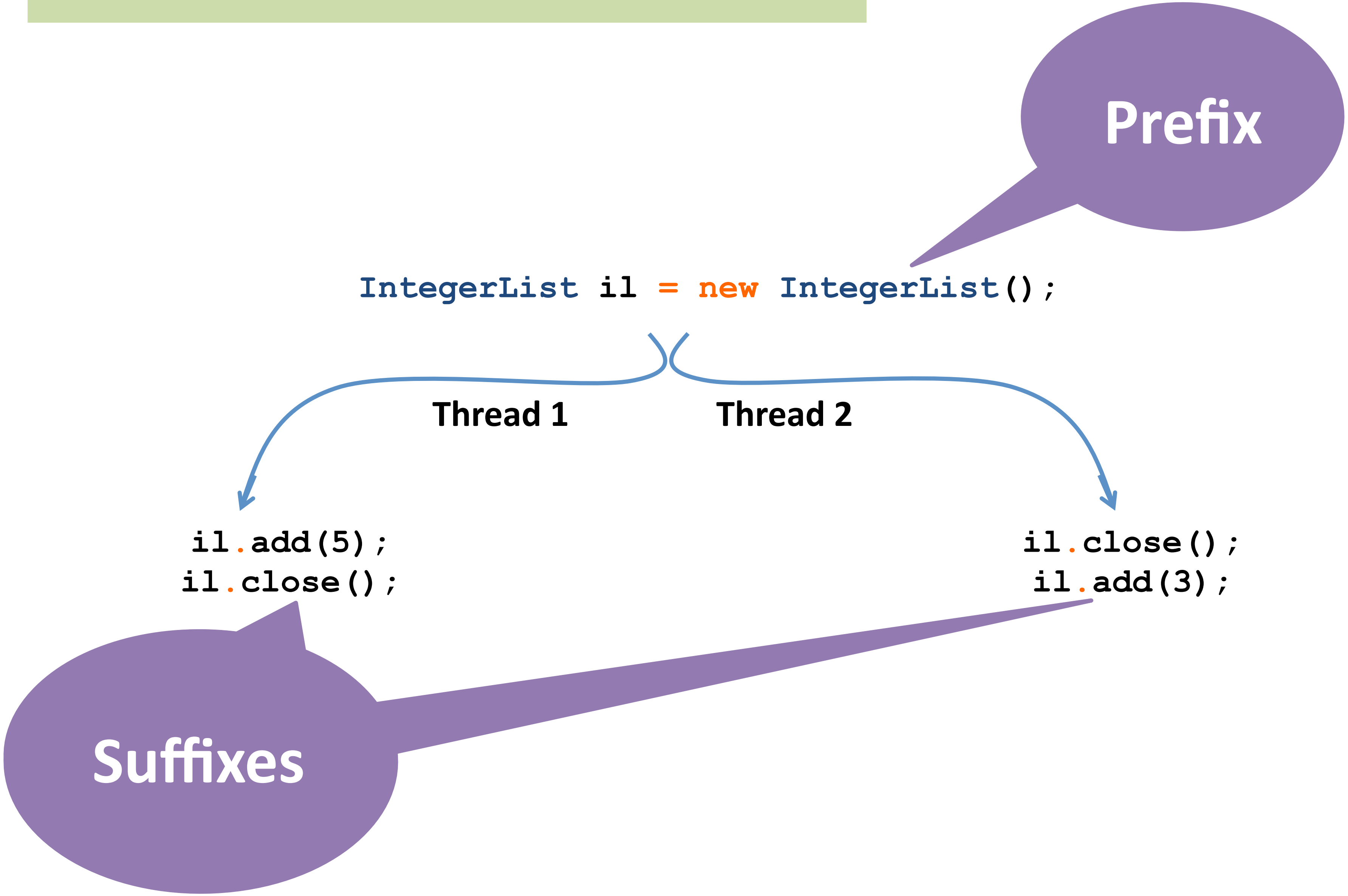
Thread 1

```
il.add(5);
il.close();
```

Thread 2

```
il.close();
il.add(3);
```

A Concurrent Test



Prefix

```
IntegerList il = new IntegerList();
```

Thread 1

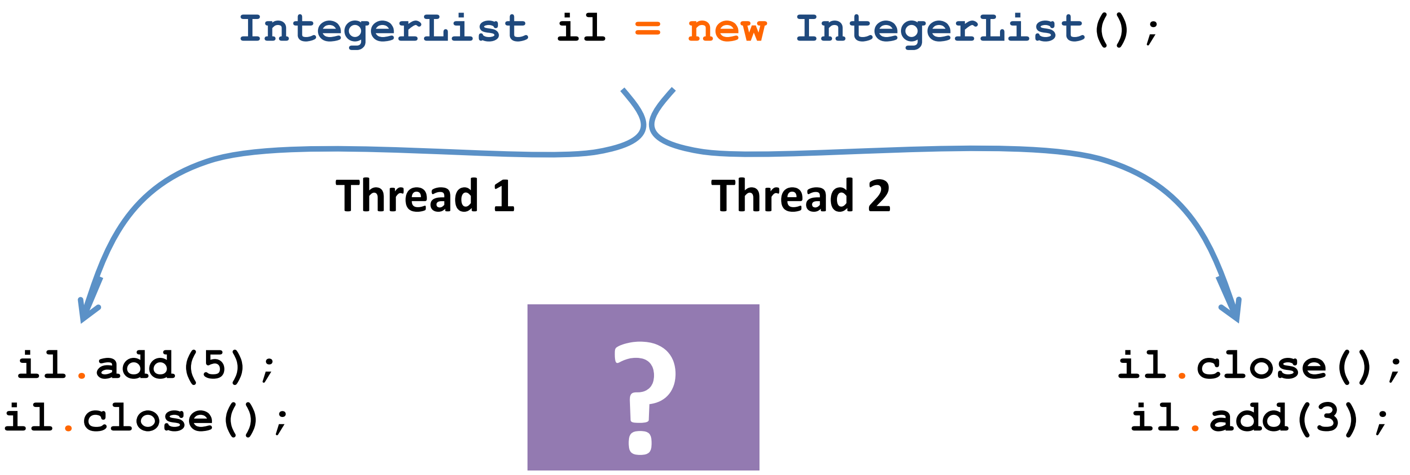
Thread 2

```
il.add(5);
il.close();
```

```
il.close();
il.add(3);
```

Suffixes

A Concurrent Test



```
IntegerList il = new IntegerList();
```

Thread 1

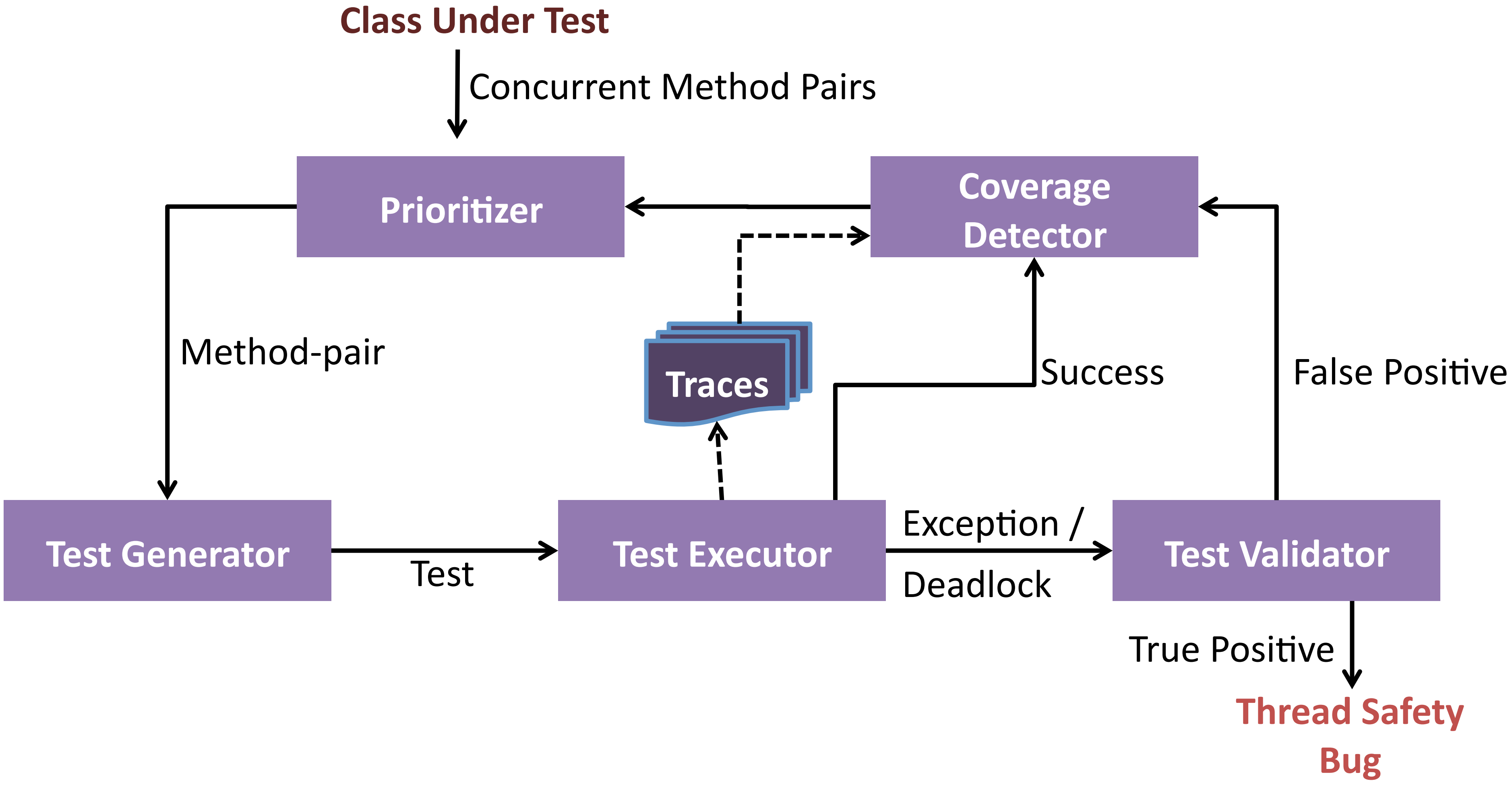
```
il.add(5);
il.close();
```

Thread 2

```
il.close();
il.add(3);
```

How do we select methods to test in suffixes ?

CovCon - Overview



Concurrent Method Pairs

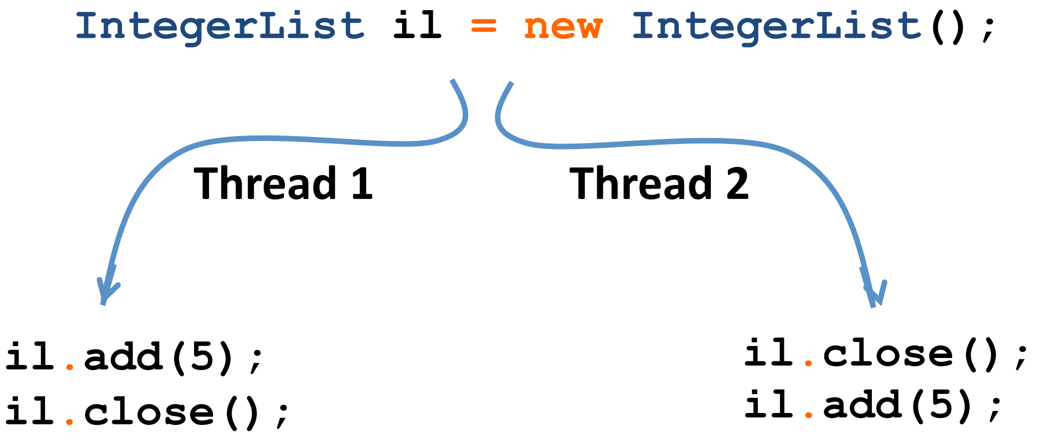
- Set of all pairs of public methods in a class and its super-class.

```
public class IntegerList {
   public synchronized void add(int num) { }
   public void close() { }
   public synchronized int getIndex(int num) { }
}
```

Method 1	Method 2
add	add
add	close
close	close
getIndex	getIndex
add	getIndex
close	getIndex

Test Generator

- Generates test using the selected method pair.



```
IntegerList il = new IntegerList();
```

Thread 1

```
il.add(5);
il.close();
```

Thread 2

```
il.close();
il.add(5);
```

Method 1	Method 2
add	add
add	close
close	close
getIndex	getIndex
add	getIndex
close	getIndex

Coverage Detector

- Analyze trace files generated in Test Executor.

Method 1	Method 2	Covered Count
add	add	0
add	close	0
close	close	0
getIndex	getIndex	0
add	getIndex	0
close	getIndex	0

Trace File 1 **Trace File 2**

Coverage Detector

- Analyze trace files generated in Test Executor.

Method 1	Method 2	Covered Count
add	add	0
add	close	0
close	close	0
getIndex	getIndex	0
add	getIndex	0
close	getIndex	0

Trace File 1

Trace File 2

```
Start:add    Time:1
```

Coverage Detector

- Analyze trace files generated in Test Executor.

Method 1	Method 2	Covered Count
add	add	0
add	close	0
close	close	0
getIndex	getIndex	0
add	getIndex	0
close	getIndex	0

Trace File 1

```
Start:add    Time:1
```

Trace File 2

```
Start:close  Time:2
```

Coverage Detector

- Analyze trace files generated in Test Executor.

Method 1	Method 2	Covered Count
add	add	0
add	close	1
close	close	0
getIndex	getIndex	0
add	getIndex	0
close	getIndex	0

Trace File 1

```
Start:add     Time:1
End:add       Time:3
```

Trace File 2

```
Start:close   Time:2
```

Coverage Detector

- Analyze trace files generated in Test Executor.

Method 1	Method 2	Covered Count
add	add	0
add	close	2
close	close	1
getIndex	getIndex	0
add	getIndex	0
close	getIndex	0

Trace File 1

```
Start:add     Time:1
End:add       Time:3
Start:close   Time:4
End:close     Time:8
```

Trace File 2

```
Start:close   Time:2
End:close     Time:5
Start:add     Time:6
End:add       Time:7
```

Prioritizer

Prioritizer

- **Tried Count (*T*):** Number of times a method-pair appears in concurrent suffixes

Prioritizer

- **Tried Count (*T*):** Number of times a method-pair appears in concurrent suffixes
- **Covered Count (*C*):** Number of times a method-pair is executed concurrently

Prioritizer

- **Tried Count (*T*):** Number of times a method-pair appears in concurrent suffixes
- **Covered Count (*C*):** Number of times a method-pair is executed concurrently
- **Coverage Score (*S*):** Lower score means higher priority

Prioritizer

- **Tried Count (*T*):** Number of times a method-pair appears in concurrent suffixes
- **Covered Count (*C*):** Number of times a method-pair is executed concurrently
- **Coverage Score (*S*):** Lower score means higher priority

$$S = \max(\mathrm{abs}(T - C), 1) * \max(T, 1)$$

A Few Executions Later ...

Lower Coverage Score = Higher Priority

Method 1	Method 2	Tried Count (*T*)	Covered Count (*C*)	Coverage Score (*S*)
`add`	`add`	6	0	36
`add`	`close`	13	11	26
`close`	`close`	8	4	32
`getIndex`	`getIndex`	6	0	36
`add`	`getIndex`	6	0	36
`close`	`getIndex`	14	12	28

$$S = \max(\mathrm{abs}(T- C), 1) * \max(T, 1)$$

A Few Executions Later ...

Method 1	Method 2	Tried Count (*T*)	Covered Count (*C*)	Coverage Score (*S*)
`add`	`add`	6	0	36
`add`	`close`	13	11	26
`close`	`close`	8	4	32
`getIndex`	`getIndex`	6	0	36
`add`	`getIndex`	6	0	36
`close`	`getIndex`	14	12	28

$$S = \max(\mathrm{abs}(T- C), 1) * \max(T, 1)$$

A Few Executions Later ...

Maybe protected by locks / synchronization

Method 1	Method 2	Tried Count (*T*)	Covered Count (*C*)	Coverage Score (*S*)
`add`	`add`	6	0	36
`add`	`close`	13	11	26
`close`	`close`	8	4	32
`getIndex`	`getIndex`	6	0	36
`add`	`getIndex`	6	0	36
`close`	`getIndex`	14	12	28

$$S = \max(\mathrm{abs}(T - C), 1) * \max(T, 1)$$

A Few Executions Later ...

Method 1	Method 2	Tried Count (*T*)	Covered Count (*C*)	Coverage Score (*S*)
`add`	`add`	6	0	36
`add`	`close`	13	11	26
`close`	`close`	8	4	32
`getIndex`	`getIndex`	6	0	36
`add`	`getIndex`	6	0	36
`close`	`getIndex`	14	12	28

$$S = \max(\text{abs}(T - C), 1) * \max(T, 1)$$

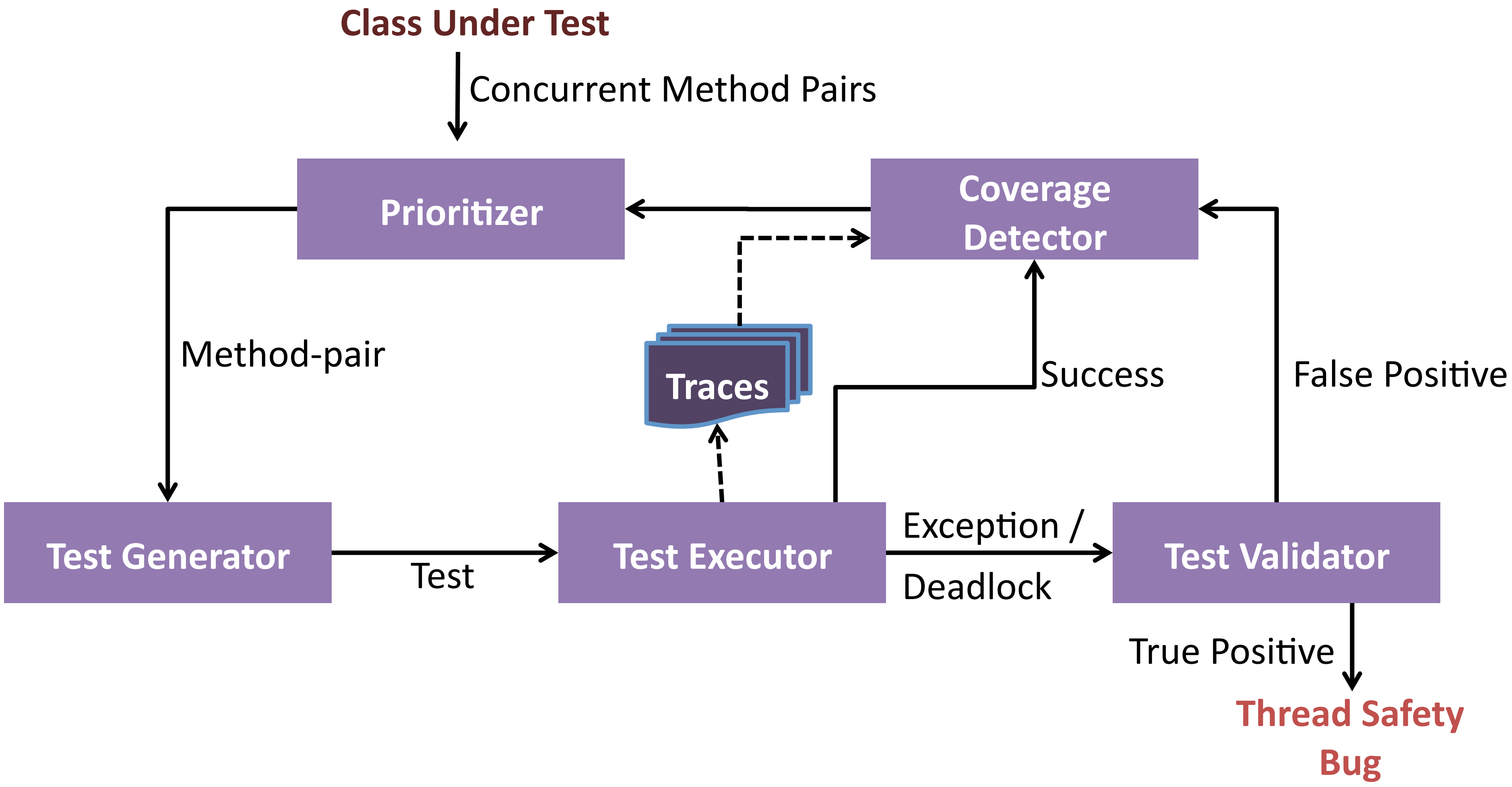
A Few Executions Later ...

Select `add` and `close`

Method 1	Method 2	Tried Count (*T*)	Covered Count (*C*)	Coverage Score (*S*)
`add`	`add`	6	0	36
`add`	`close`	13	11	26
`close`	`close`	8	4	32
`getIndex`	`getIndex`	6	0	36
`add`	`getIndex`	6	0	36
`close`	`getIndex`	14	12	28

S = max(abs(T- C), 1) * max(T, 1)

Executor and Validator



Fully Automatic and Precise Detection of Thread Safety Violations. Michael Pradel and Thomas R. Gross (PLDI 2012).

Evaluation - Setup

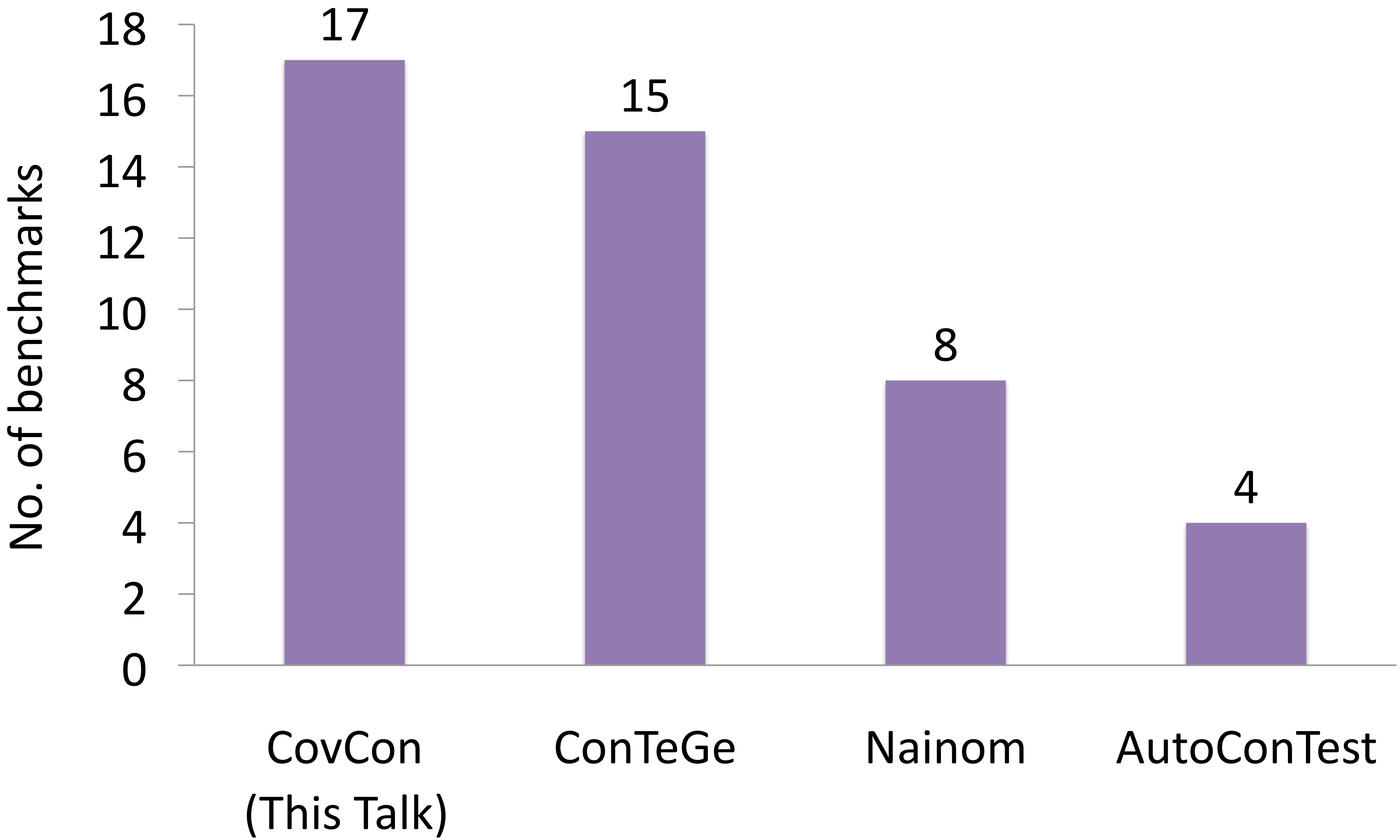
- 18 thread-safe classes (StringBuffer, Vector, XStream, etc).

- Each benchmark is executed 10 times.

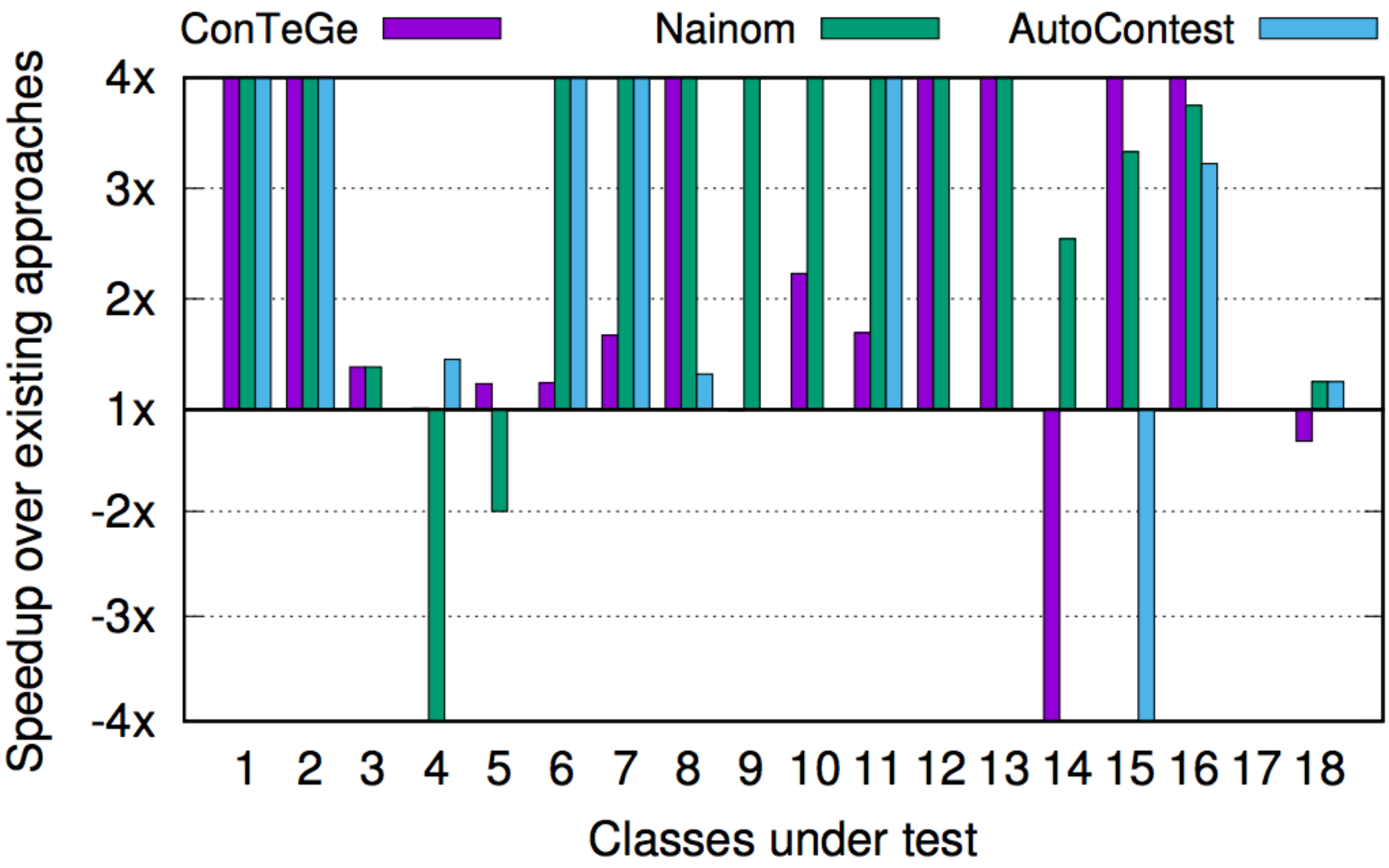
- Timeout: 1 hour for each execution of a benchmark.

- Approaches evaluated:
  - CovCon$^{\text{ICSE ’17}}$: Coverage-based Approach (this talk).
  - ConTeGe$^{\text{PLDI ’12}}$: Random-based Approach.
  - Nainom$^{\text{OOPSLA ’14; FSE ’15; PLDI ’15}}$: Sequential Tests based Approach.
  - AutoConTest$^{\text{ICSE ’16}}$: Coverage-based Approach.

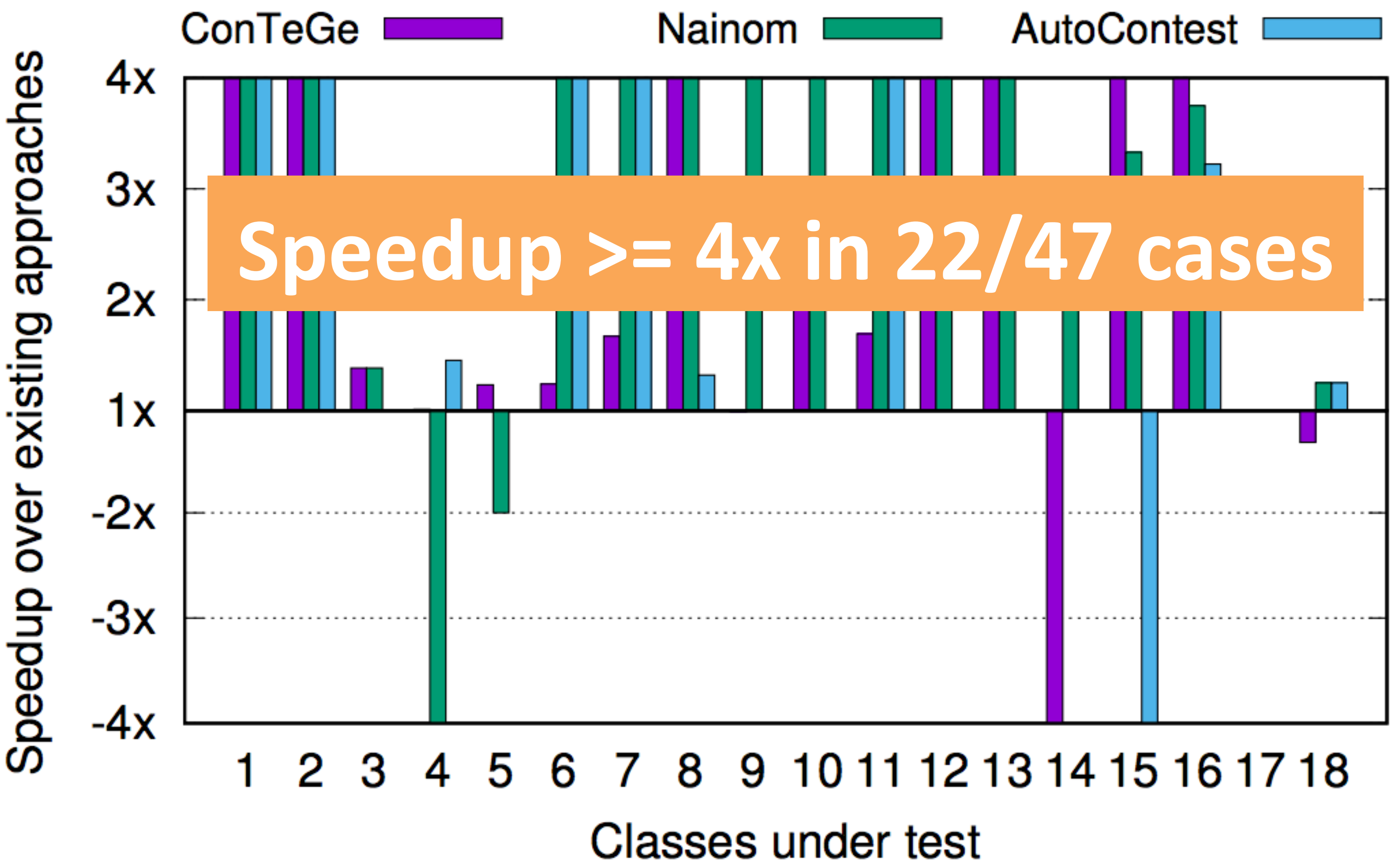
Bug Finding Capability



Speedup: Time to Find Bug

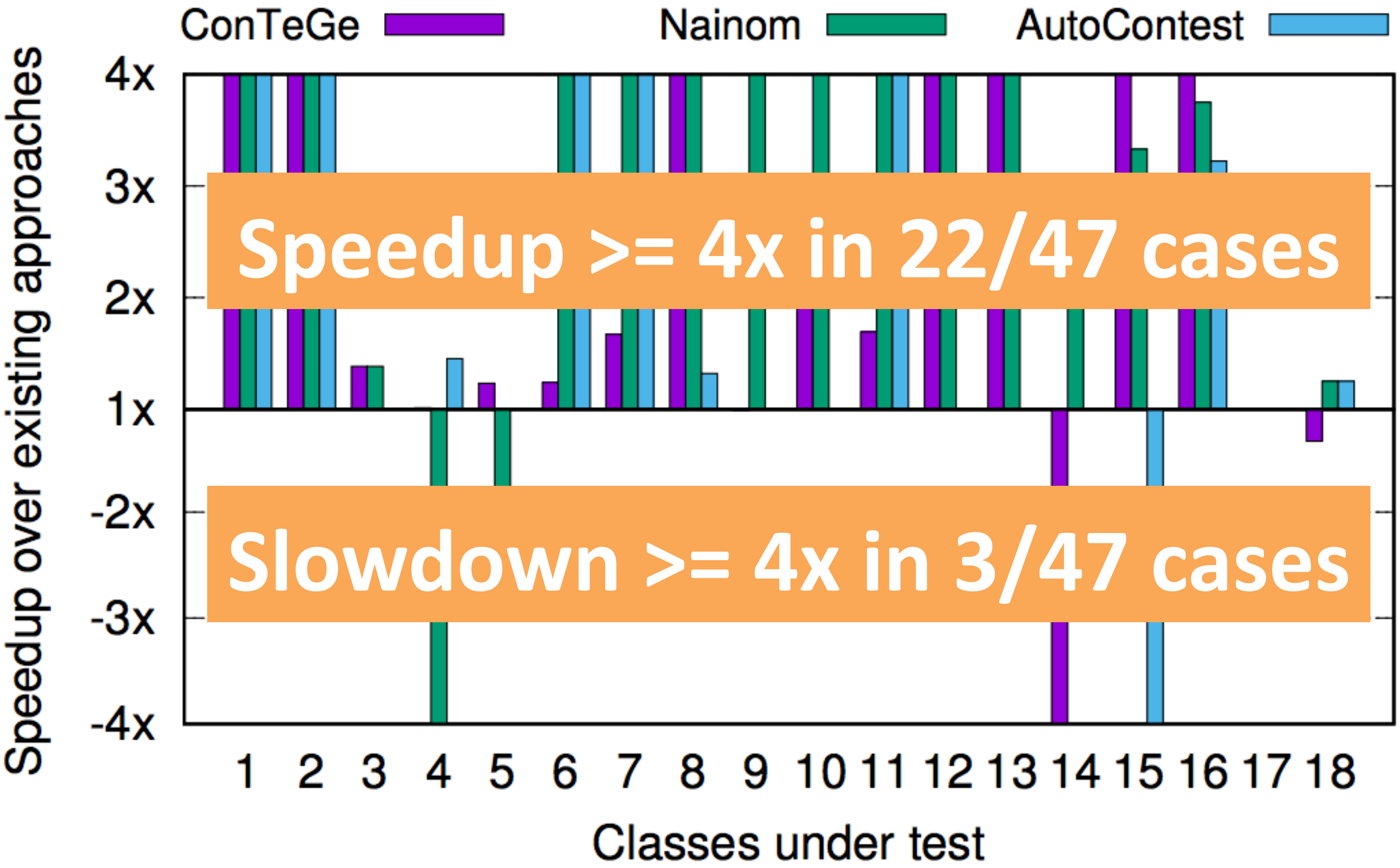


Speedup: Time to Find Bug



Speedup >= 4x in 22/47 cases

Speedup: Time to Find Bug



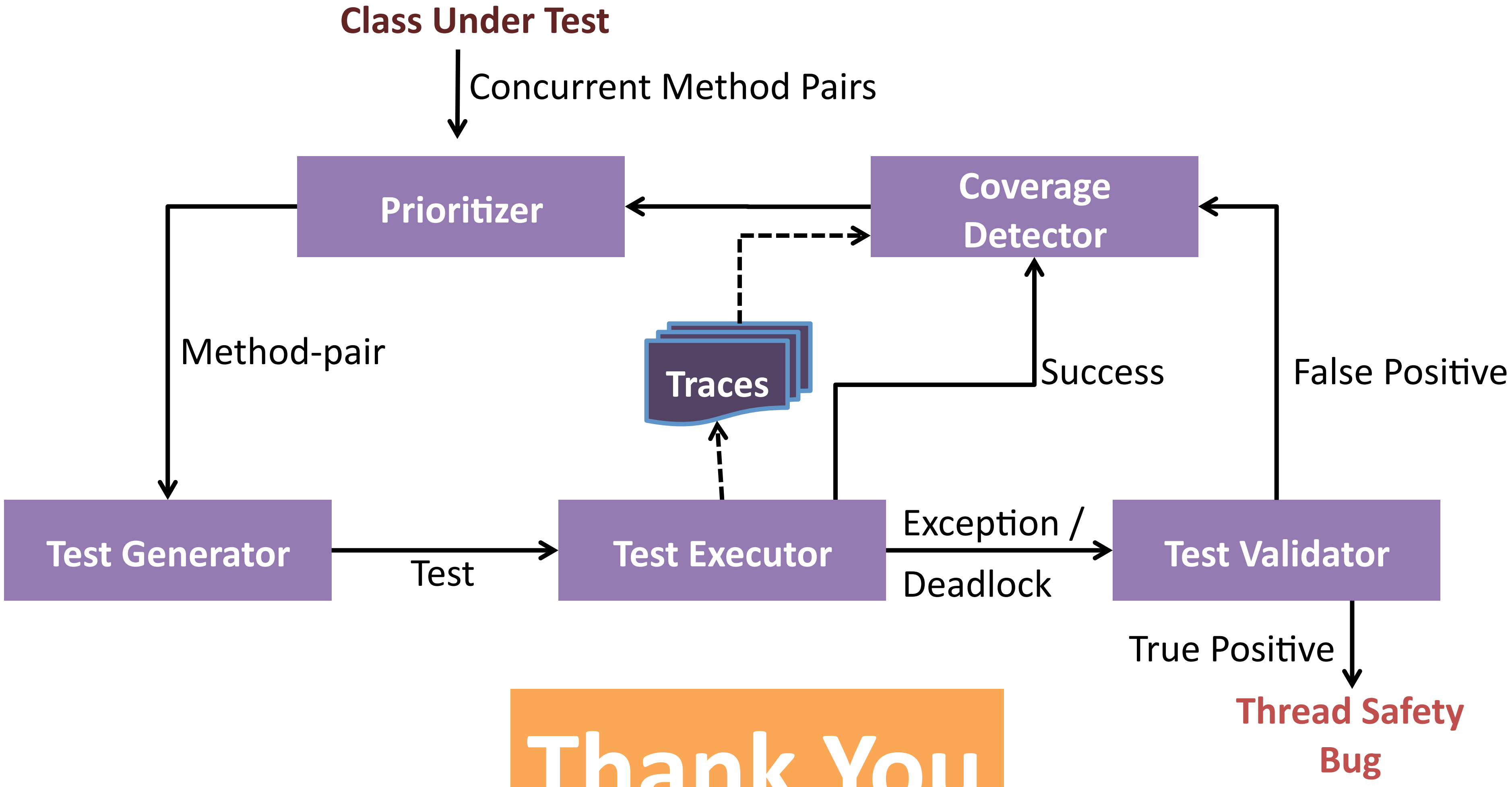
Speedup >= 4x in 22/47 cases

Slowdown >= 4x in 3/47 cases

Conclusion

- Simple. Effective. Efficient.
- Inexpensive coverage analysis.
- Tests generated towards infrequently covered method pairs.
- Dynamically assigns lower priority to method pairs which are synchronized/lock protected.

CovCon - Overview

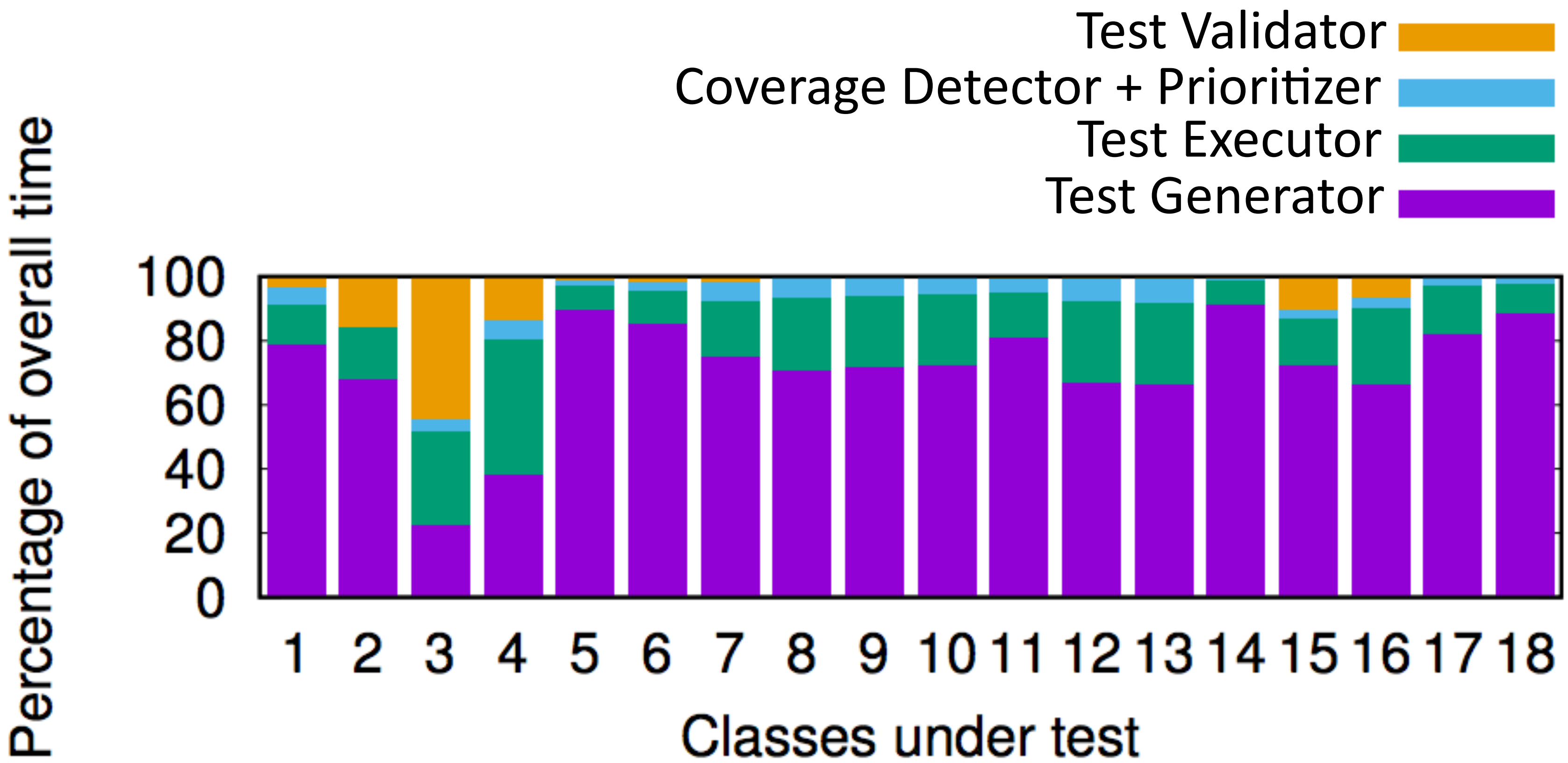


Thank You

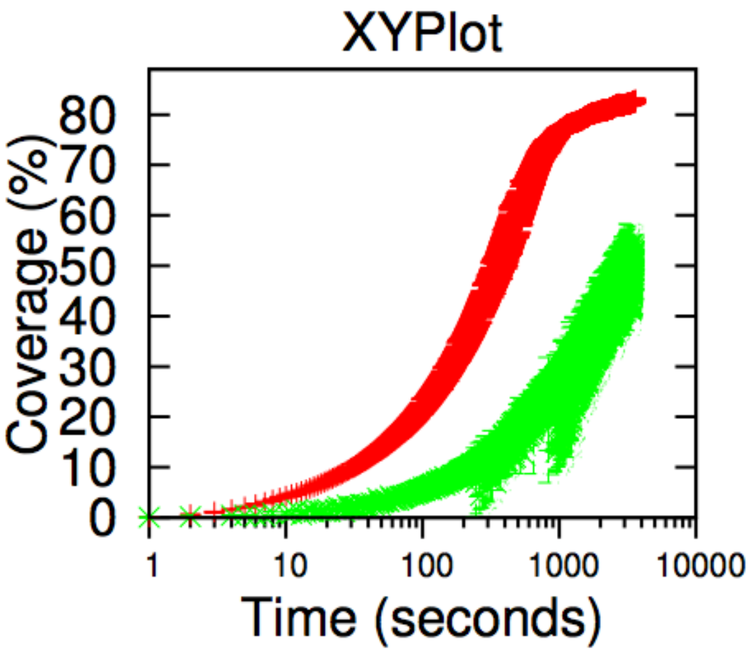
Benchmarks

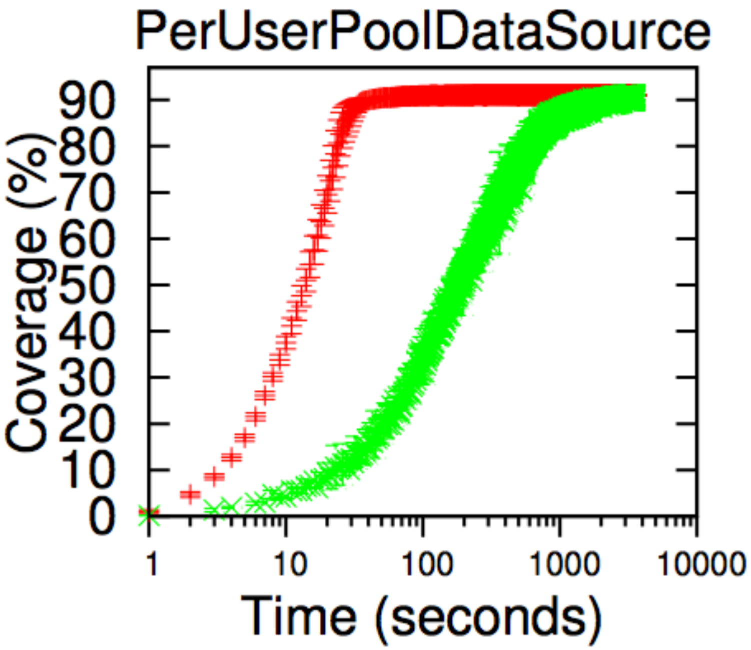
Id	Class	Code Base	Type of Bug	Methods (#)	CMPs (#)
1	BufferedInputStream	JDK 1.1	Atomicity Violation	9	45
2	Logger	JDK 1.4.1	Atomicity Violation	44	990
3	SynchronizedMap	JDK 1.4.2	Deadlock	15	120
4	ConcurrentHashMap	JDK 1.6.0	Atomicity Violation	22	253
5	StringBuffer	JDK 1.6.0	Atomicity Violation	52	1378
6	TimeSeries	JFreeChart 0.9.8	Race Condition	41	861
7	XYSeries	JFreeChart 0.9.8	Race Condition	25	325
8	NumberAxis	JFreeChart 0.9.12	Atomicity Violation	110	6105
9	PeriodAxis	JFreeChart 1.0.1	Race Condition	125	7875
10	XYPlot	JFreeChart 1.0.9	Race Condition	217	23653
11	Day	JFreeChart 1.0.13	Race Condition	26	351
12	PerUserPoolDataSource	CommonsDBCP 1.4	Race Condition	65	2145
13	SharedPoolDataSource	CommonsDBCP 1.4	Race Condition	51	1326
14	XStream	XStream 1.4.1	Race Condition	66	2211
15	Vector	JDK 1.1.7	Atomicity Violation	24	300
16	Vector	JDK 1.4.2	Atomicity Violation	45	1035
17	IntRange	Apache Commons 2.4	Atomicity Violation	26	351
18	AsMap	Google Commons 1.0	Atomicity Violation	15	120

Coverage Measurement Cost



Coverage-driven


XYPlot
Coverage (%)
Time (seconds)


PerUserPoolDataSource
Coverage (%)
Time (seconds)