Università della Svizzera italiana

ICST 2019

UNIVERSITÀ DEGLI STUDI DI MILANO BICOCCA

Coverage-Driven Test Generation for Thread-Safe Classes via Parallel and Conflict Dependencies

IEEE TCSE Distinguished Paper Award

Valerio Terragni*

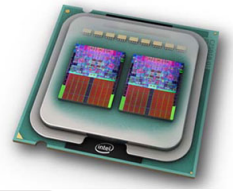
Mauro Pezzè*◇

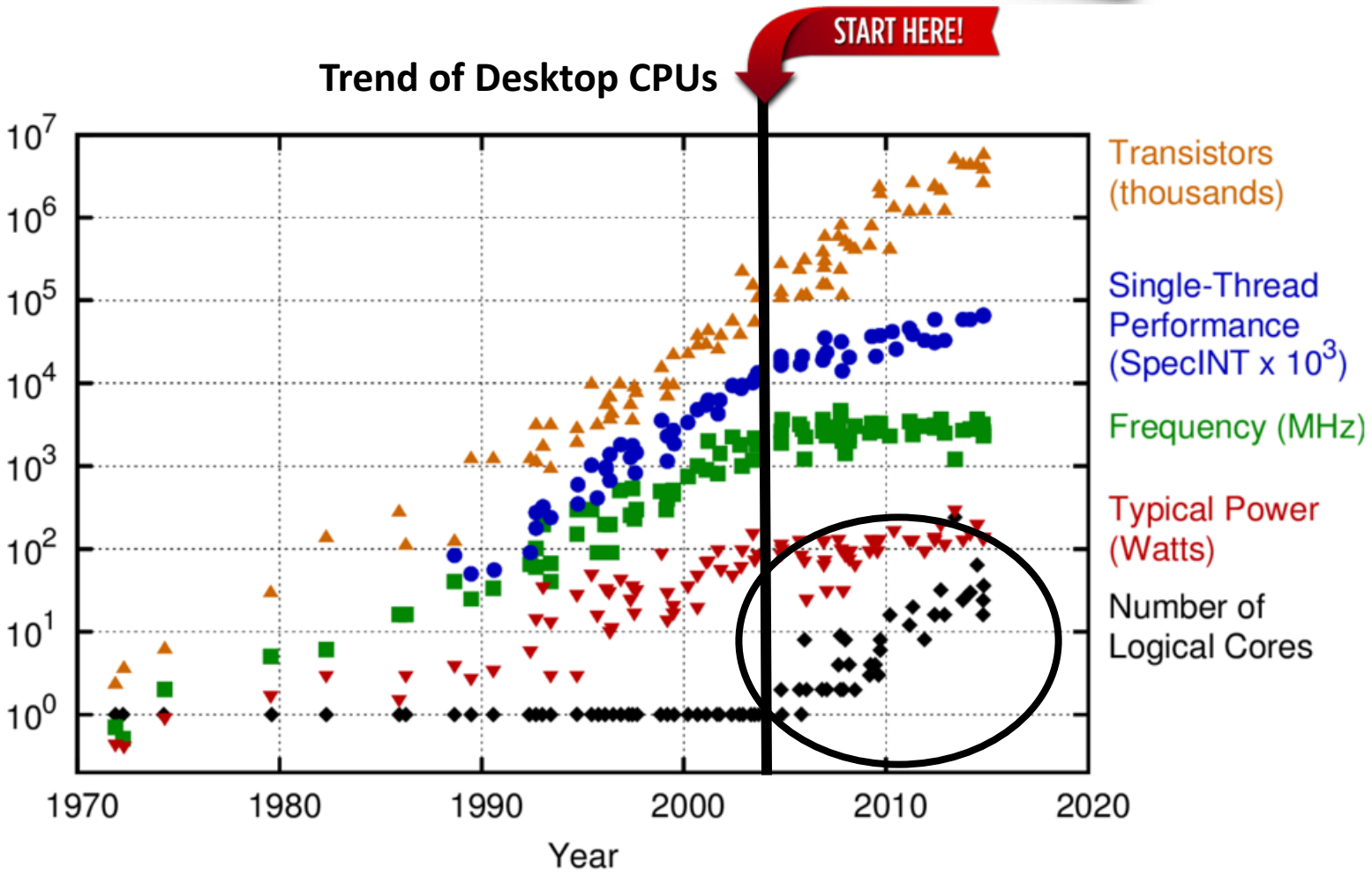
Francesco Bianchi*

* USI Università della Svizzera italiana
Switzerland

◇ Università di Milano Bicocca
Italy

Multi-Core Era

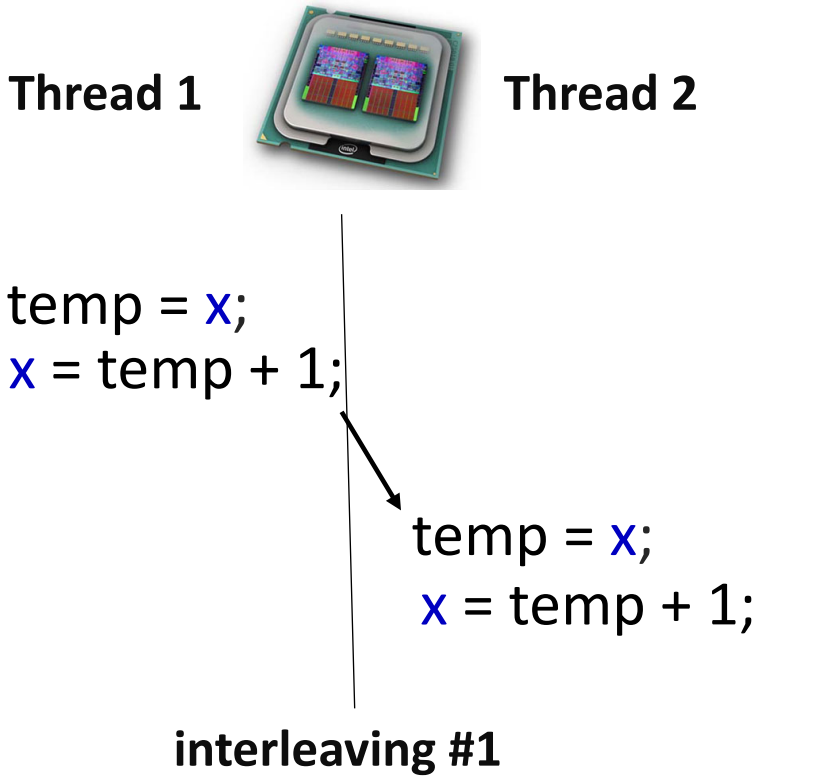


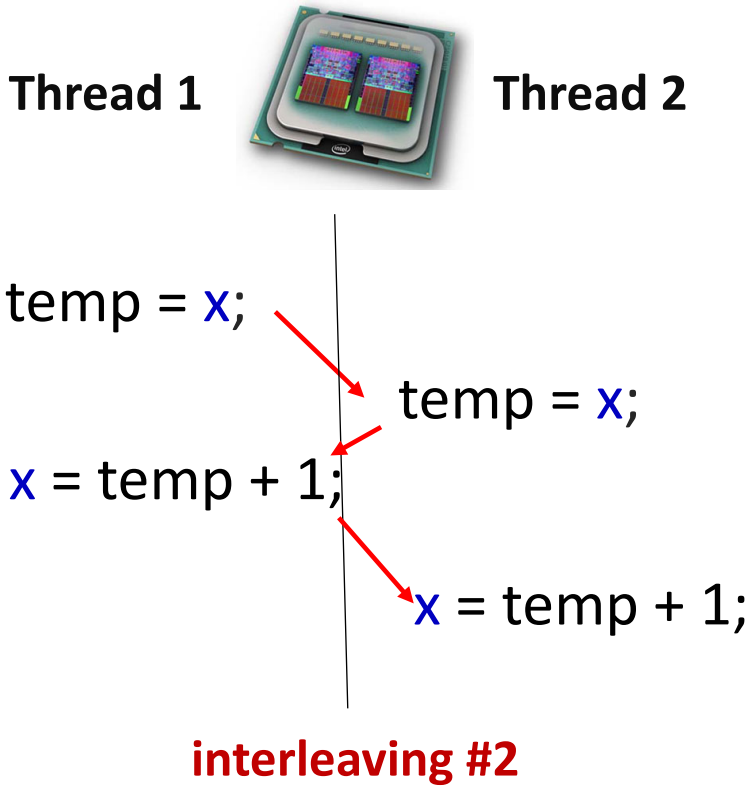


Plot from : https://goo.gl/MJALxM

Non-Deterministic Thread Interleavings

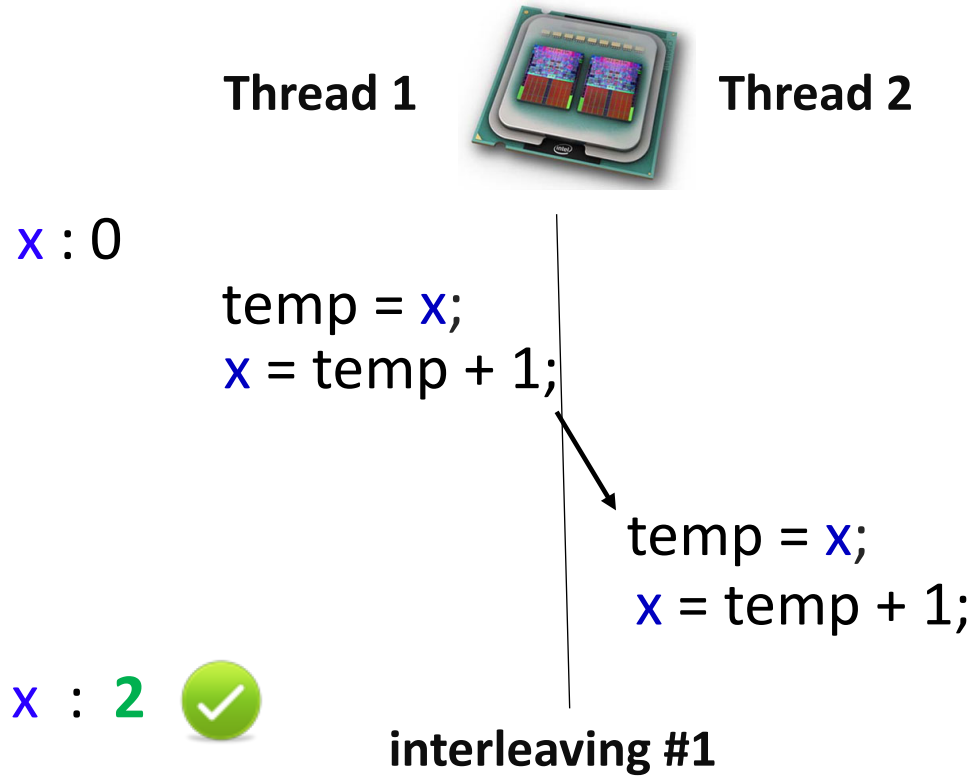
Execution orders of shared-memory accesses among threads

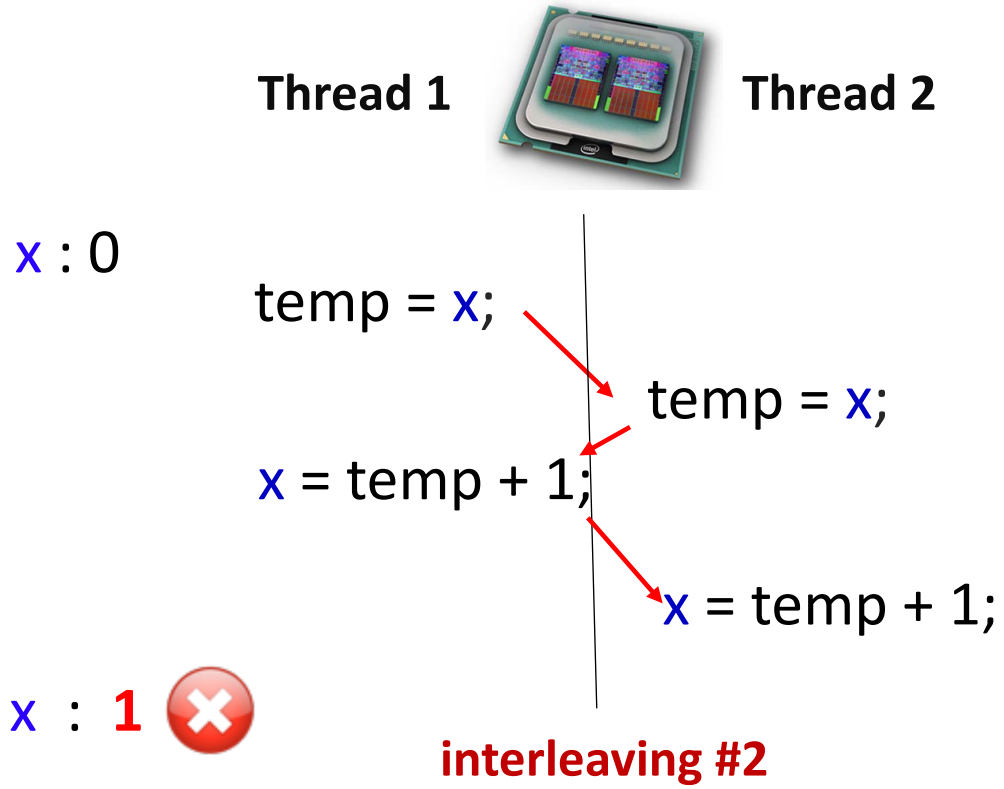




Non-Deterministic Thread Interleavings

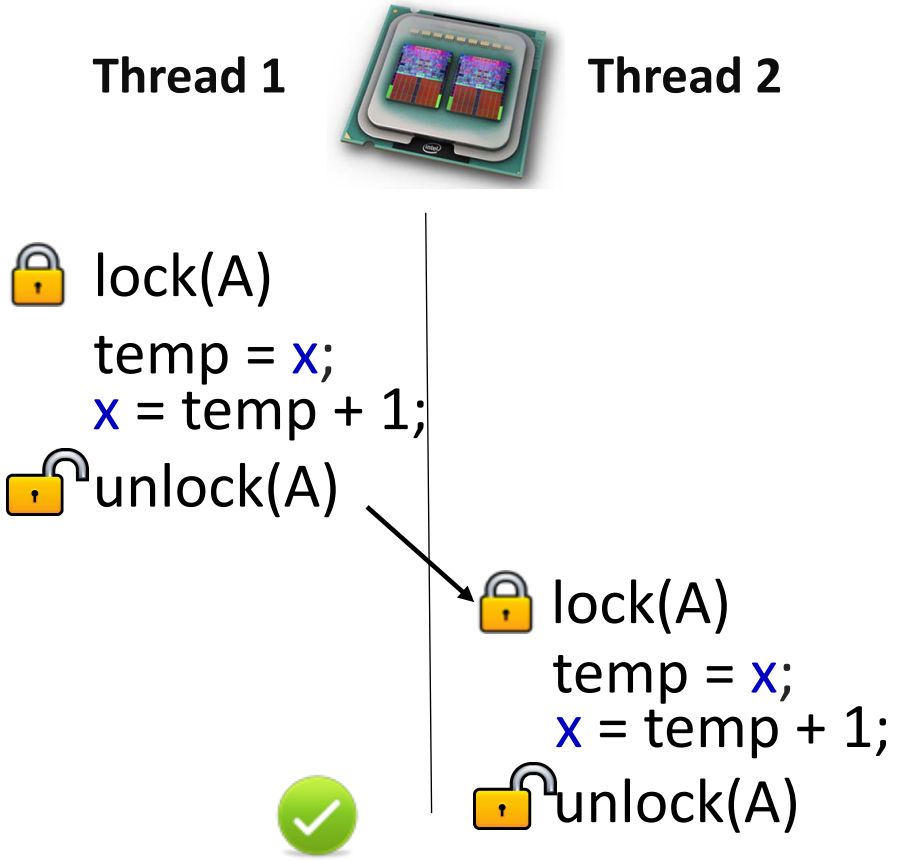
Execution orders of shared-memory accesses among threads

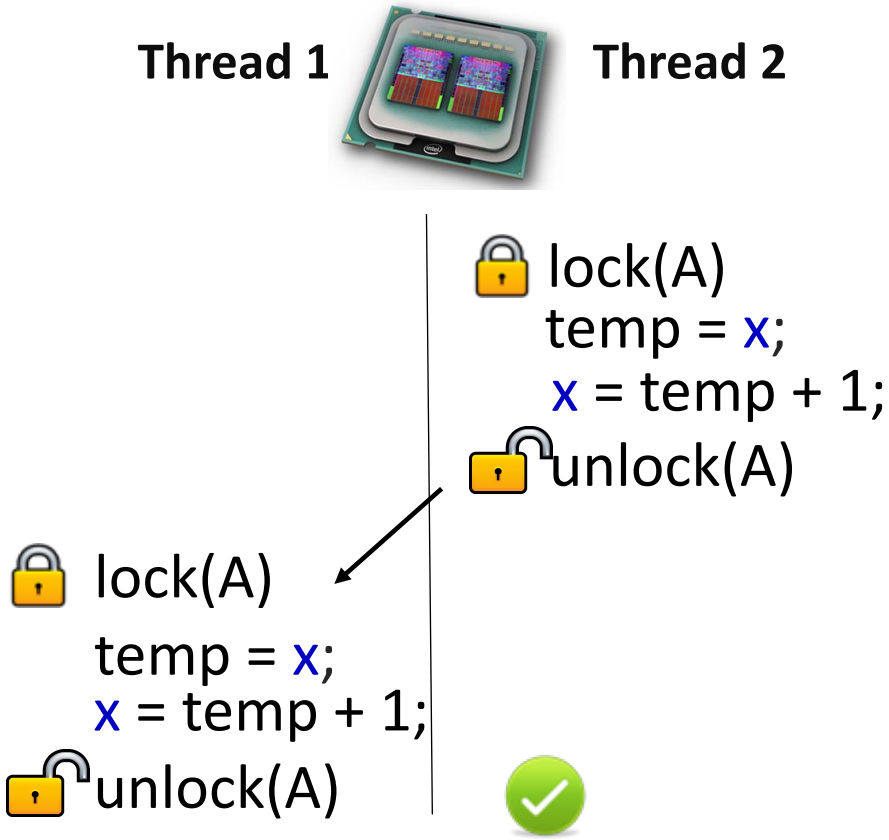




Thread Synchronization

e.g., lock and unlock operations





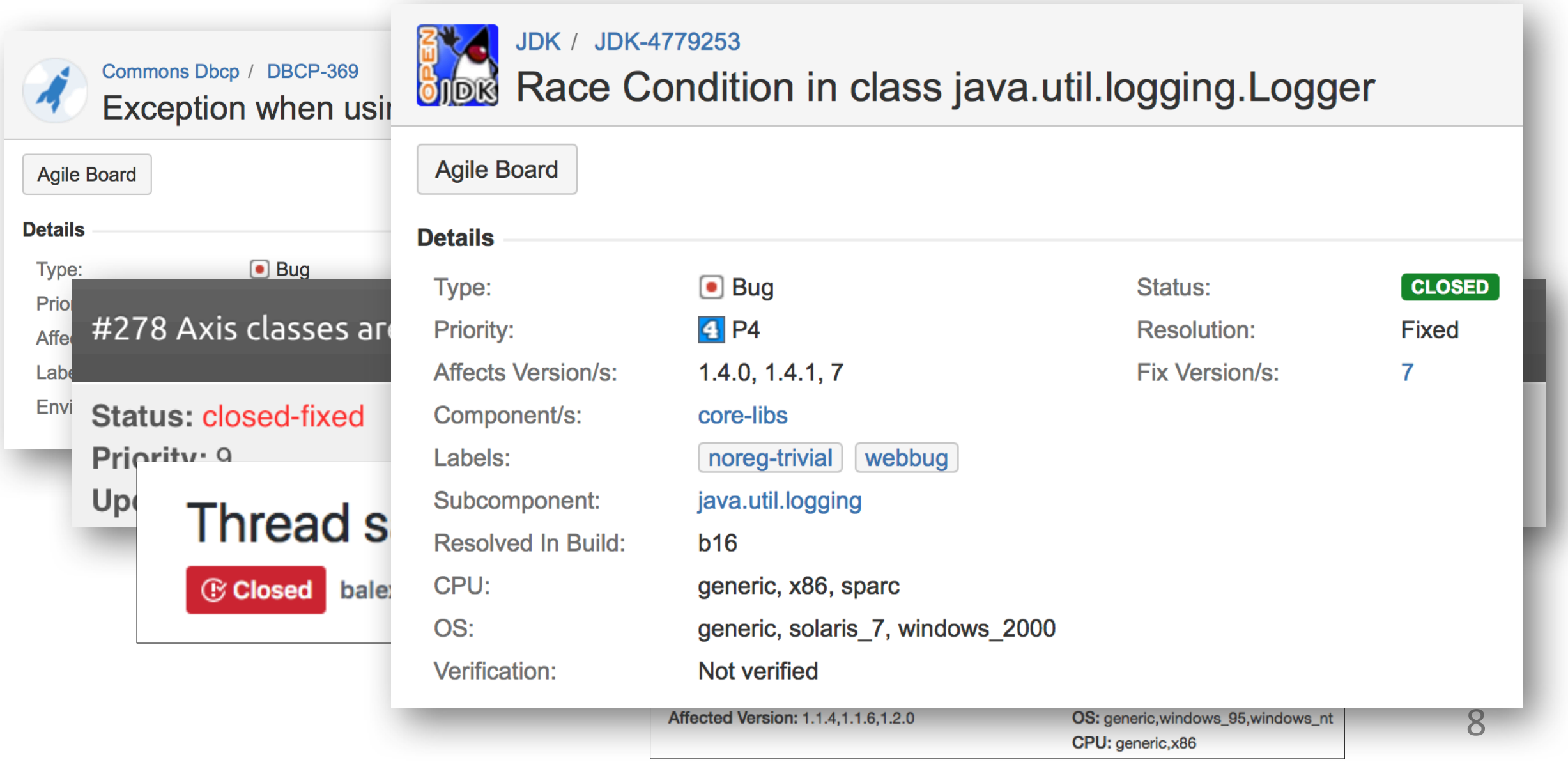
Synchronization is Challenging

Thread-safe Classes

"A class that encapsulates synchronizations that ensure a correct behavior when the same instance of the class is accessed from multiple threads"


```
public class C1 {
 private int x;
 private int y;

 public C1() { … }

 public synchronized void m1(int k, C2 a) {….}

 public void m2() {
   …
   synchronized(this){…}
   …
 }
```

Thread-Safe Classes are Buggy



Thread-Safe Classes are Buggy

ORACLE Java Bug Database

Oracle Technology Network > Java > Java SE > Community > Bug Database

JDK-4728096 : java.io.BufferedInputStream has no synchronization on close operation

Type: Bug
Component: core-libs
Sub-Component: java.io
Affected Version: 1.4.0,1.4.1,1.4.2

Priority: P4
Status: Resolved
Resolution: Fixed
OS: generic,linux,linux_redhat_6.1, ...
CPU: generic,x86,sparc

Submi
Updat
Resolv

Thread-Safety Violation (Example)

JDK-4728096 : java.io.BufferedInputStream has no synchronization on close operation

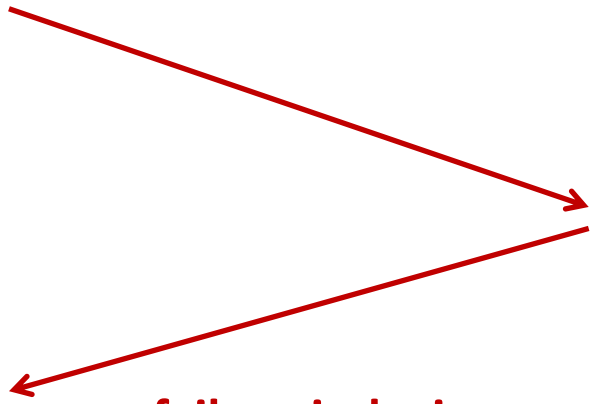
Thread 1

```
public synchronized int read() {
 ensureOpen();
 if (pos >= count) {
   fill();
   if (pos >= count)
       return -1;
  }
 return buf[pos++] & 0xff;
 }
```

NullPointer Exception

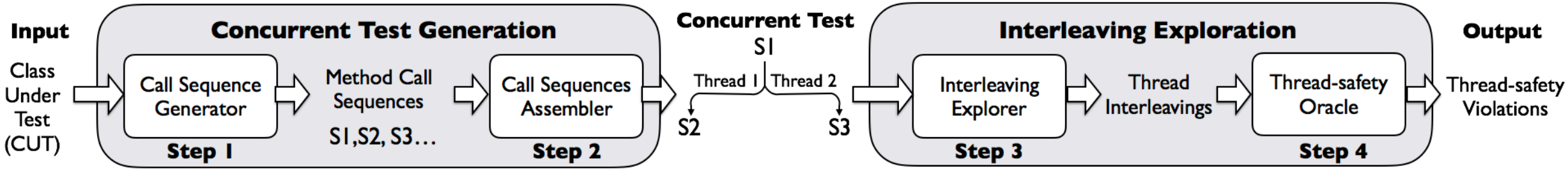
Thread 2

```
// missing synchronization
public void close() {
   if (in == null)
       return;
   in.close();
   in = null;
   buf = null;
    }
```



failure-inducing thread interleaving

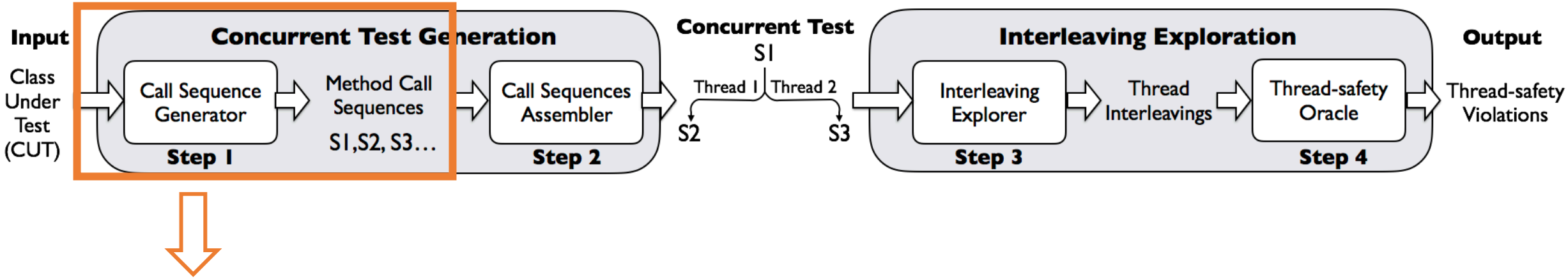
Automated Concurrent Test Generation



General Framework

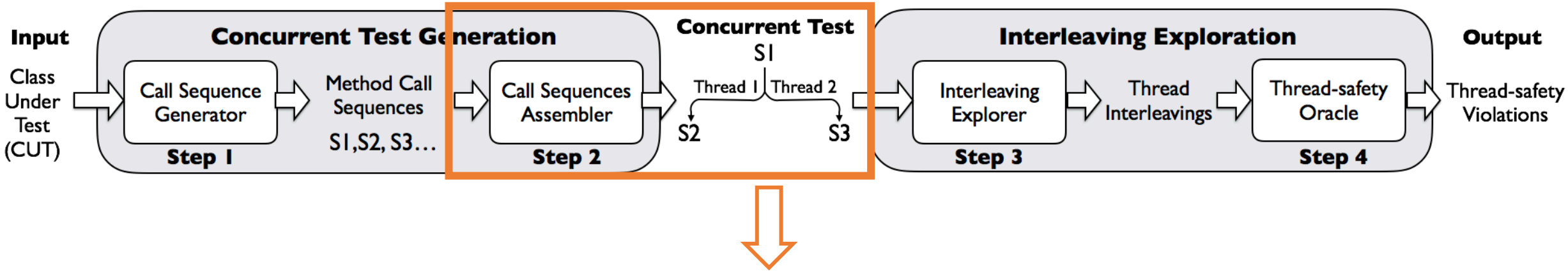
Valerio Terragni and Mauro Pezzè
Effectiveness and Challenges in Generating Concurrent Tests for Thread-Safe Classes.
ASE 2018

Automated Concurrent Test Generation

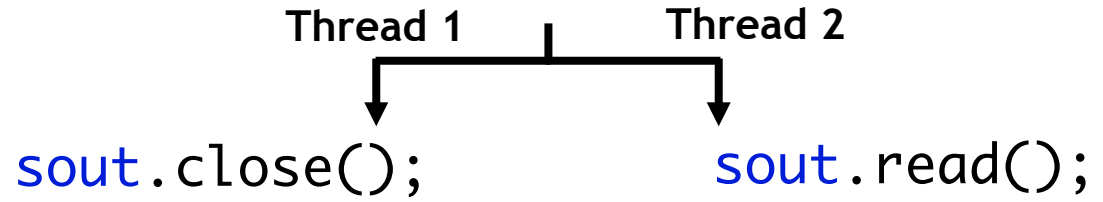



```
StringBufferInputStream var0 = new StringBufferInputStream("v;");
BufferedInputStream sout = new BufferedInputStream(var0);
sout.close();

….

StringBufferInputStream var0 = new StringBufferInputStream("v;");
BufferedInputStream sout = new BufferedInputStream(var0);
sout.read();
```

Automated Concurrent Test Generation

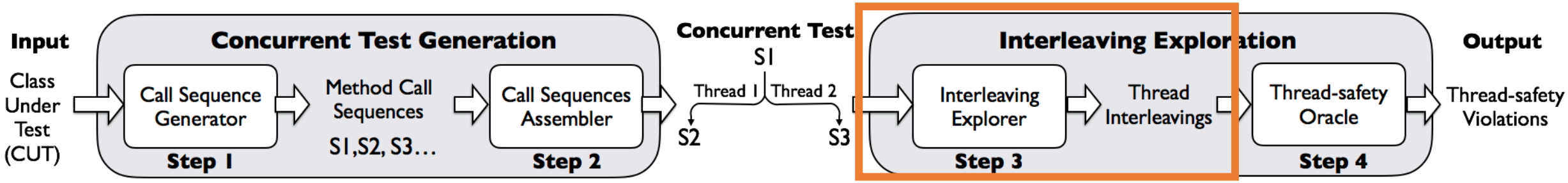



```
StringBufferInputStream var0 = new StringBufferInputStream("v;");
BufferedInputStream sout = new BufferedInputStream(var0);
```

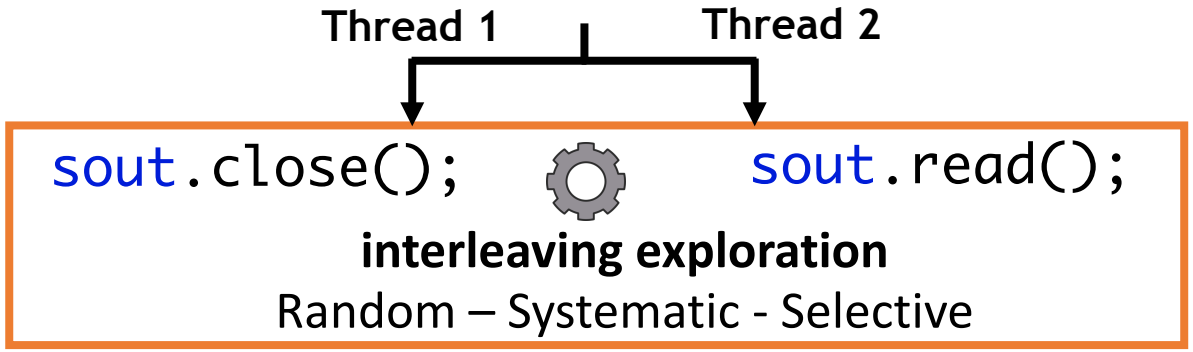


Set of method call sequences that exercise the public interface of a class from multiple threads

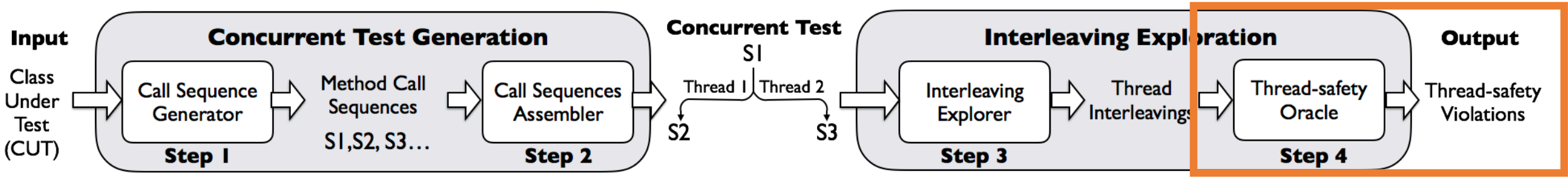
Automated Concurrent Test Generation




```
StringBufferInputStream var0 = new StringBufferInputStream("v;");
BufferedInputStream sout = new BufferedInputStream(var0);
```



Automated Concurrent Test Generation




```
if (pos >= count)
    ….
```

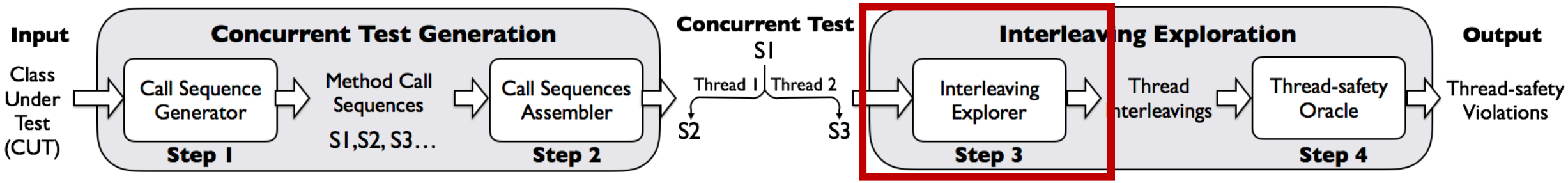
```
buf = null;
```

```
buf[pos++] & 0xff;
```

NullPointer Exception

Thread Safety Oracle: **Linearizability** (Herlihy@TOPLAS `90)

Challenges



Challenges

 # 1 – Step 3 Interleaving exploration is expensive!

Implication: we cannot generate and explore the interleaving space of many concurrent tests

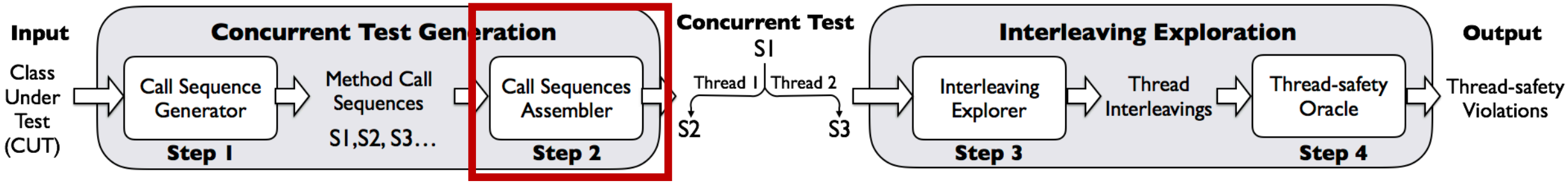
$$\frac{(N_1+N_2)!}{N_1!\ N_2!}$$ # possible interleavings **$9.2 \cdot 10^{128}$**

2 threads

50 N_1= #shared memory accesses thread 1

50 N_2= #shared memory accesses thread 2

Challenges



Challenges

 # 1 – Step 3 Interleaving exploration is expensive!

Implication: we cannot generate and explore the interleaving space of many concurrent tests

 # 2 – Step 2 Huge space of concurrent tests!

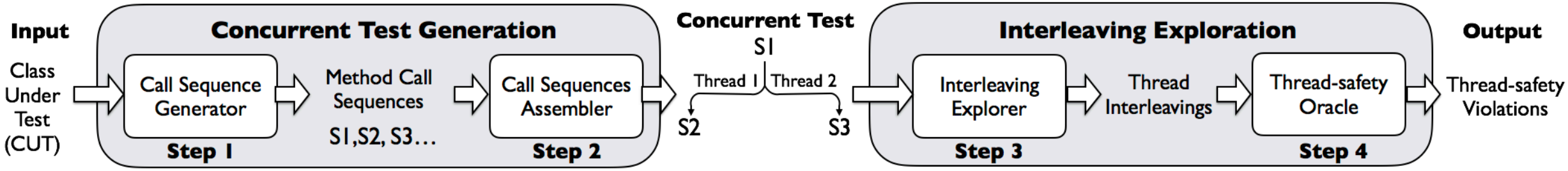
BufferedInpuStream

# of possible concurrent tests = M^{TL} 10^{30}

M = # methods, L = max length of method call sequence, T = # threads (>= 3)

10 **10** **3**

Challenges

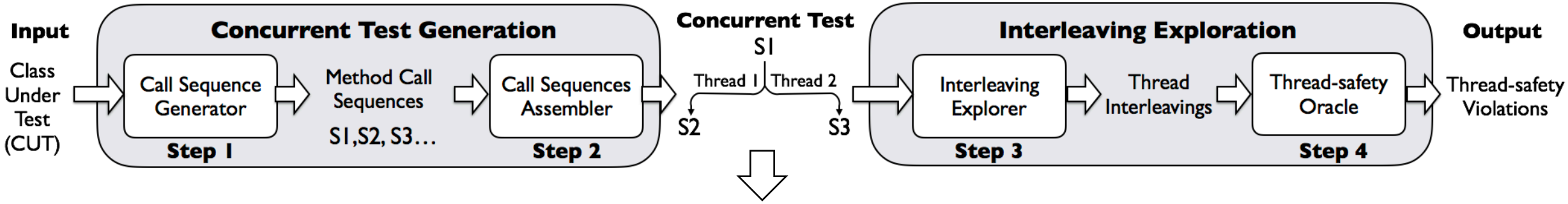


Challenges

 # 1 – Step 3 Interleaving exploration is expensive!

 # 2 – Step 2 Huge space of concurrent tests!

How can we generate fewer tests that are likely to expose thread-safety violations?



Avoid generating concurrent tests that are **redundant**

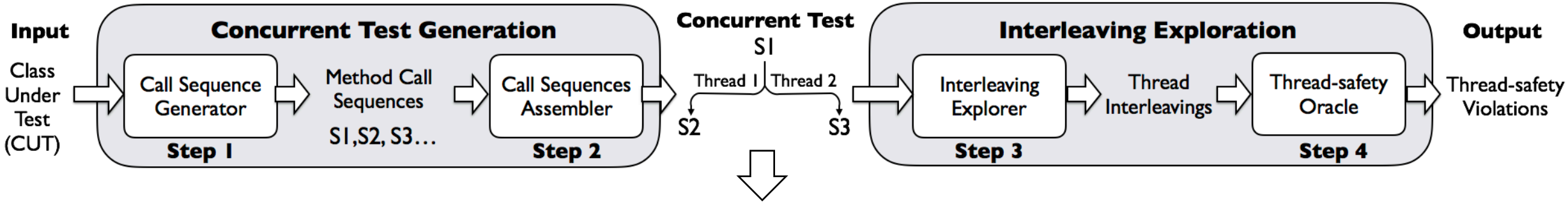
Coverage-driven concurrent test generation

Choudhary @ ICSE 2017

Terragni @ ICSE 2016

Steenbuck @ ICST 2013

Our Intuition



Avoid generating concurrent tests that are redundant **and irrelevant** for exposing thread-safety violations

Coverage-driven concurrent test generation

Choudhary @ ICSE 2017

Terragni @ ICSE 2016

Steenbuck @ ICST 2013

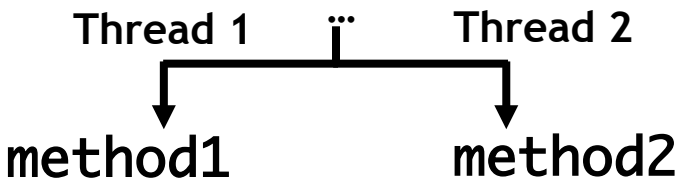
conflict and parallel dependencies

NEW!

DepCon

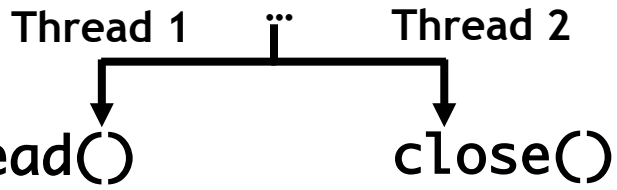
Conflict Dependency: `method1` and `method2` access at least one same shared-memory location (W-R, R-W)

Parallel Dependency: the execution `method1` and `method2` can interleave



Conflict Dependency: `read()` and `close()` access at least one same shared-memory location (W-R, R-W)

Parallel Dependency: the execution `read()` and `close()` can interleave



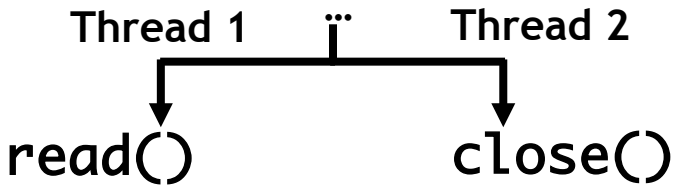
EXAMPLE 1

```
public synchronized int read() {
 ensureOpen();
 if (pos >= count) {
   fill();
   if (pos >= count)
       return -1;
  }
 return buf[pos++] & 0xff;
 }
```

```
public void close() {
  if (in == null)
      return;
  in.close();
  in = null;
  buf = null;
   }
```

Conflict Dependency: `read()` and `close()` access at least one same shared-memory location (W-R, R-W)

Parallel Dependency: the execution `read()` and `close()` can interleave

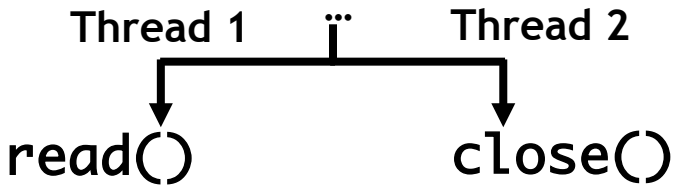



```
public synchronized int read() {
 ensureOpen();
 if (pos >= count) {
   fill();
   if (pos >= count)
       return -1;
  }
 return buf[pos++] & 0xff;
 }
```

```
public void close() {
  if (in == null)
      return;
  in.close();
  in = null;
  buf = null;
  }
```

Conflict Dependency: `read()` and `close()` access at least one same shared-memory location (W-R, R-W)

Parallel Dependency: the execution `read()` and `close()` can interleave

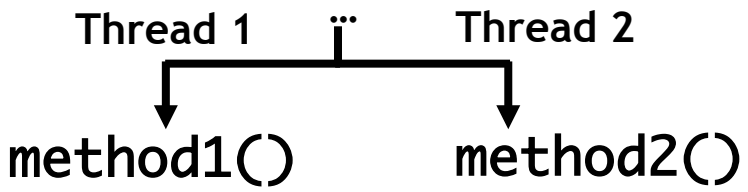



```
public synchronized int read() {
 ensureOpen();
 if (pos >= count) {
   fill();
   if (pos >= count)
       return -1;
  }
 return buf[pos++] & 0xff;
 }
```

```
public void close() {
   if (in == null)
       return;
   in.close();
   in = null;
   buf = null;
    }
```

Conflict Dependency: `method1` and `method2` access at least one same shared-memory location (W-R, R-W)

Parallel Dependency: the execution `method1` and `method2` can interleave

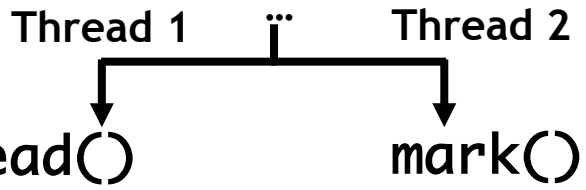


Theorem 1
Having both conflict and parallel dependencies is a **necessary condition** for exposing a thread-safe violation

Conflict Dependency: `read()` and `mark()` access at least one same shared-memory location (W-R, R-W)

Parallel Dependency: the execution `read()` and `mark()` can interleave

EXAMPLE 2

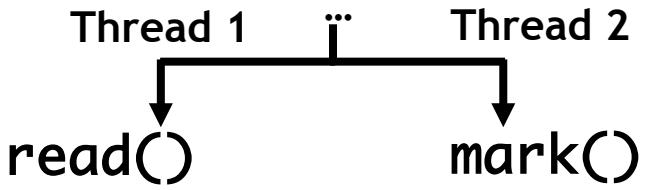



```
public synchronized int read() {
 ensureOpen();
 if (pos >= count) {
   fill();
   if (pos >= count)
       return -1;
  }
 return buf[pos++] & 0xff;
 }
```

```
public synchronized void mark(int readlimit) {
  marklimit = readlimit;
  markpos = pos;
}
```

Conflict Dependency: `read()` and `mark()` access at least one same shared-memory location (W-R, R-W)

Parallel Dependency: the execution `read()` and `mark()` can interleave

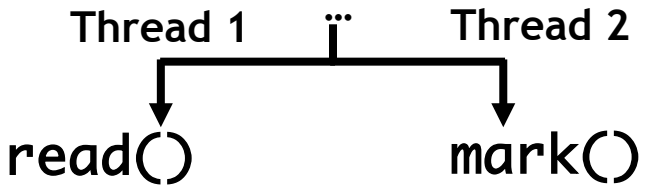



```
public synchronized int read() {
 ensureOpen();
 if (pos >= count) {
   fill();
   if (pos >= count)
       return -1;
  }
 return buf[pos++] & 0xff;
 }
```

```
public synchronized void mark(int readlimit) {
  marklimit = readlimit;
  markpos = pos;
}
```

Conflict Dependency: `read()` and `mark()` access at least one same shared-memory location (W-R, R-W)

Parallel Dependency: the execution `read()` and `mark()` can interleave




```
public synchronized int read() {
 ensureOpen();
 if (pos >= count) {
   fill();
   if (pos >= count)
       return -1;
 }
 return buf[pos++] & 0xff;
 }
```

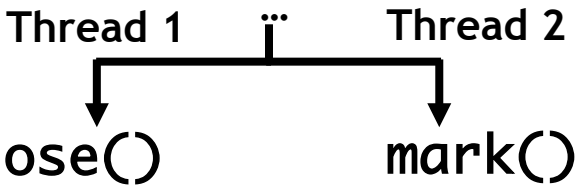
```
public synchronized void mark(int readlimit) {
   marklimit = readlimit;
   markpos = pos;
}
```

DepCon will NOT generate concurrent tests that execute `read()` and `mark()` concurrently

Conflict Dependency: `close()` and `mark()` access at least one same shared-memory location (W-R, R-W)

Parallel Dependency: the execution `close()` and `mark()` can interleave

EXAMPLE 3

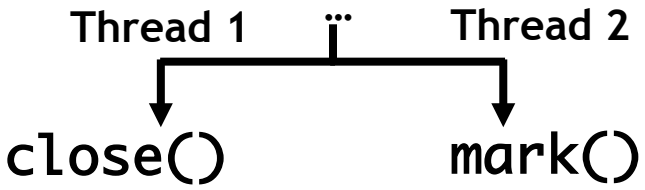



```
public void close() {
  if (in == null)
      return;
  in.close();
  in = null;
  buf = null;
  }
```

```
public synchronized void mark(int readlimit) {
  marklimit = readlimit;
  markpos = pos;
}
```

Conflict Dependency: `close()` and `mark()` access at least one same shared-memory location (W-R, R-W)

Parallel Dependency: the execution `close()` and `mark()` can interleave

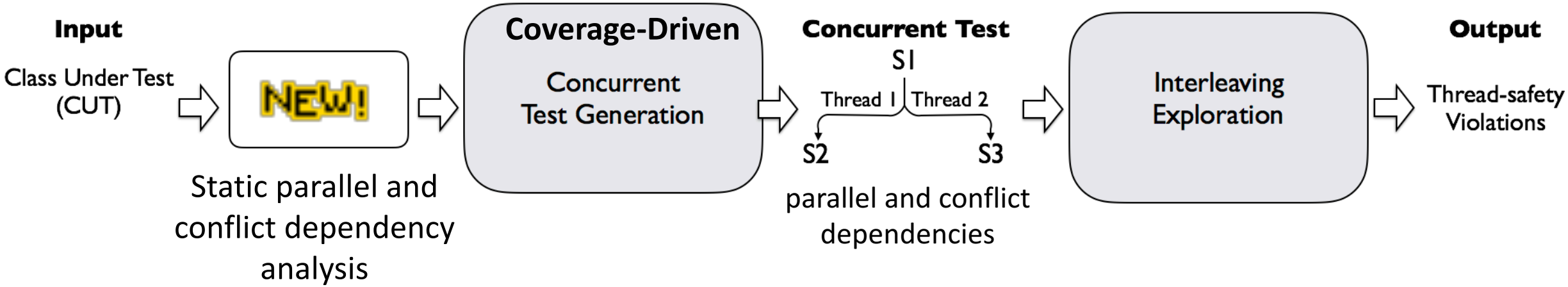



```
public void close() {
  if (in == null)
      return;
  in.close();
  in = null;
  buf = null;
  }
```

```
public synchronized void mark(int readlimit) {
  marklimit = readlimit;
  markpos = pos;
}
```

DepCon will NOT generate concurrent tests that execute `close()` and `mark()` concurrently

DepCon


Input
Class Under Test (CUT)
NEW!
Static parallel and conflict dependency analysis
Coverage-Driven
Concurrent Test Generation
Concurrent Test
S1
Thread 1
Thread 2
S2
S3
parallel and conflict dependencies
Interleaving Exploration
Output
Thread-safety Violations

Computing the Dependencies

Method summaries

ACCESS SUMMARY : it represents an over-approximation of all the possible accesses of shared-memory locations

{R(in), R(count), R(pos)}

LOCK SUMMARY : *set of locks that always protect every shared-memory accesses*

{this}

```
private void ensureOpen() {
  if (in == null)
    throw new IOException("Stream closed");
}
```

```
public synchronized int available() {
  ensureOpen();
  return (count - pos) + in.available();
}
```

Computing the Dependencies

Challenges

- Efficiency (overhead should be low)
- Completeness (no missed dependencies)
- High precision (effective search space pruning)

Solution

- Novel and effective combination of classic static analysis techniques

lockset analysis

inter-procedural analysis

purity analysis

alias analysis

```
public synchronized int read() {
ensureOpen();
 if (pos >= count) {
   fill();
   if (pos >= count)
       return -1;
 }
return buf[pos++] & 0xff;
}
```

Evaluation

RQ1 Effectiveness

Can DepCon effectively generate concurrent tests that expose thread-safety violations?

RQ2 Comparison

Is DepCon more effective than state- of-the-art concurrent test generation?

RQ3 Static Analysis

What is the efficiency, completeness and precision of DepCon's Static Analysis?

Code Base	Class Name	LOC	# public methods	fault type
Apache Math	IntRange	276	26	Atomicity violation
Apache DBCP	PerUserPoolDataSource	719	66	Data race
	SharedPoolDataSource	546	52	Atomicity violation
HQSQLDB	DoubleIntIndex	966	34	Atomicity violation
JDK	BufferedInputStream	239	10	Atomicity violation
	Vector	786	45	Atomicity violation
JFreeChart	Day	267	26	Data race
	NumberAxis	1,662	111	Atomicity violation
	PeriodAxis	1,975	126	Data race
	TimeSeries	359	41	Data race
	XYPlot	3,080	218	Data race
	XYSeries	200	25	Data race
Log4J	FileAppender	369	21	Atomicity violation
	WriterAppender	317	24	Atomicity violation
XStream	XStream	926	66	Data race

Subjects

- 7 popular Java code bases
- 15 known concurrency bugs
- Subjects used in the evaluation of previous work

RQ1 Effectiveness

Class name	LOC
IntRange	276
PerUserPoolDataSource	719
SharedPoolDataSource	546
DoubleIntIndex	966
BufferedInputStream	239
Vector	786
Day	267
NumberAxis	1,662
PeriodAxis	1,975
TimeSeries	359
XYPlot	3,080
XYSeries	200
FileAppender	369
WriterAppender	317
XStream	926
AVG	

- Build on top of CovCon (Choudhary@ICSE2017)
- Interleaving Explorer: Stress testing (100 iterations)
- Time budget of 1 hour
- 5 runs

RQ1 Effectiveness

Class name	LOC	Success Rate
IntRange	276	100%
PerUserPoolDataSource	719	40%
SharedPoolDataSource	546	100%
DoubleIntIndex	966	60%
BufferedInputStream	239	100%
Vector	786	20%
Day	267	100%
NumberAxis	1,662	40%
PeriodAxis	1,975	100%
TimeSeries	359	100%
XYPlot	3,080	40%
XYSeries	200	100%
FileAppender	369	20%
WriterAppender	317	20%
XStream	926	80%
AVG		**68%**

- Build on top of CovCon (Choudhary@ICSE2017)
- Interleaving Explorer: Stress testing (100 iterations)
- Time budget of 1 hour
- 5 runs

RQ1 Effectiveness

Class name	LOC	Success Rate	AVG Detection Time (hh:mm:ss)	
IntRange	276	100%	00:01:21	
PerUserPoolDataSource	719	40%	00:43:57	
SharedPoolDataSource	546	100%	00:11:51	
DoubleIntIndex	966	60%	00:34:13	
BufferedInputStream	239	100%	00:00:07	**MIN**
Vector	786	20%	00:51:11	
Day	267	100%	00:01:24	
NumberAxis	1,662	40%	00:39:50	
PeriodAxis	1,975	100%	00:03:15	
TimeSeries	359	100%	00:02:19	
XYPlot	3,080	40%	00:43:54	
XYSeries	200	100%	00:00:33	
FileAppender	369	20%	00:48:18	**MAX**
WriterAppender	317	20%	00:48:06	
XStream	926	80%	00:11:44	
AVG		**68%**	**00:22:48**	

- Build on top of CovCon (Choudhary@ICSE2017)
- Interleaving Explorer: Stress testing (100 iterations)
- Time budget of 1 hour
- 5 runs

RQ1 Effectiveness

Class name	LOC	Success Rate	AVG Detection Time (hh:mm:ss)	AVG # Generated Tests	
IntRange	276	100%	00:01:21	188	
PerUserPoolDataSource	719	40%	00:43:57	2,810	
SharedPoolDataSource	546	100%	00:11:51	1,221	
DoubleIntIndex	966	60%	00:34:13	2,617	
BufferedInputStream	239	100%	00:00:07	18	MIN
Vector	786	20%	00:51:11	1,174	
Day	267	100%	00:01:24	255	
NumberAxis	1,662	40%	00:39:50	5,860	
PeriodAxis	1,975	100%	00:03:15	438	
TimeSeries	359	100%	00:02:19	370	
XYPlot	3,080	40%	00:43:54	4,113	MAX
XYSeries	200	100%	00:00:33	116	
FileAppender	369	20%	00:48:18	2,622	
WriterAppender	317	20%	00:48:06	2,609	
XStream	926	80%	00:11:44	222	
AVG		**68%**	**00:22:48**	**1,642**	

- Build on top of CovCon (Choudhary@ICSE2017)
- Interleaving Explorer: Stress testing (100 iterations)
- Time budget of 1 hour
- 5 runs

Class name	DepCon (this work) Success Rate	DepCon (this work) AVG Detection Time (hh:mm:ss)	DepCon (this work) # Generated Tests	CovCon (Choudhary@ICSE2017) Success Rate	CovCon (Choudhary@ICSE2017) AVG Detection Time (hh:mm:ss)	CovCon (Choudhary@ICSE2017) # Generated Tests
IntRange	100%	00:01:21	188	40%	00:36:15	6,660
PerUserPoolDataSource	40%	00:43:57	2,810	40%	00:52:36	9,055
SharedPoolDataSource	100%	00:11:51	1,221	20%	00:56:00	8,947
DoubleIntIndex	60%	00:34:13	2,617	100%	00:10:38	1,055
BufferedInputStream	100%	00:00:07	18	100%	00:00:34	161
Vector	20%	00:51:11	1,174	20%	00:54:52	5,010
Day	100%	00:01:24	255	100%	00:03:03	640
NumberAxis	40%	00:39:50	5,860	0%	01:00:00	12,368
PeriodAxis	100%	00:03:15	438	100%	00:09:52	1,720
TimeSeries	100%	00:02:19	370	100%	00:15:56	2,895
XYPlot	40%	00:43:54	4,113	20%	00:57:56	4,320
XYSeries	100%	00:00:33	116	100%	00:07:47	1,842
FileAppender	20%	00:48:18	2,622	0%	01:00:00	16,264
WriterAppender	20%	00:48:06	2,609	0%	01:00:00	15,911
XStream	80%	00:11:44	222	80%	00:26:54	568
AVG	**68%**	**00:22:48**	**1,642**	**40%**	**00:34:10**	**5,828**

Class name	DepCon (this work) Success Rate		AVG Detection Time (hh:mm:ss)		# Generated Tests		CovCon (Choudhary@ICSE2017) Success Rate	AVG Detection Time (hh:mm:ss)	# Generated Tests
IntRange	100%	+60%	00:01:21	26.72x	188	35.35x	40%	00:36:15	6,660
PerUserPoolDataSource	40%		00:43:57	1.20x	2,810	3.22x	40%	00:52:36	9,055
SharedPoolDataSource	100%	+80%	00:11:51	4.73x	1,221	7.33x	20%	00:56:00	8,947
DoubleIntIndex	60%	-40%	00:34:13	0.31x	2,617	0.40x	100%	00:10:38	1,055
BufferedInputStream	100%		00:00:07	5.21x	18	8.77x	100%	00:00:34	161
Vector	20%		00:51:11	1.07x	1,174	4.27x	20%	00:54:52	5,010
Day	100%		00:01:24	2.17x	255	2.51x	100%	00:03:03	640
NumberAxis	40%	+40%	00:39:50	1.51x	5,860	2.11x	0%	01:00:00	12,368
PeriodAxis	100%		00:03:15	3.04x	438	3.93x	100%	00:09:52	1,720
TimeSeries	100%		00:02:19	6.88x	370	7.82x	100%	00:15:56	2,895
XYPlot	40%	+20%	00:43:54	1.32x	4,113	1.05x	20%	00:57:56	4,320
XYSeries	100%		00:00:33	14.05x	116	15.88x	100%	00:07:47	1,842
FileAppender	20%	+20%	00:48:18	1.24x	2,622	6.20x	0%	01:00:00	16,264
WriterAppender	20%	+20%	00:48:06	1.25x	2,609	6.10x	0%	01:00:00	15,911
XStream	80%		00:11:44	2.29x	222	2.56x	80%	00:26:54	568
AVG	**68%**	**+13%**	**00:22:48**	**4.87x**	**1,642**	**7.17x**	**40%**	**00:34:10**	**5,828**

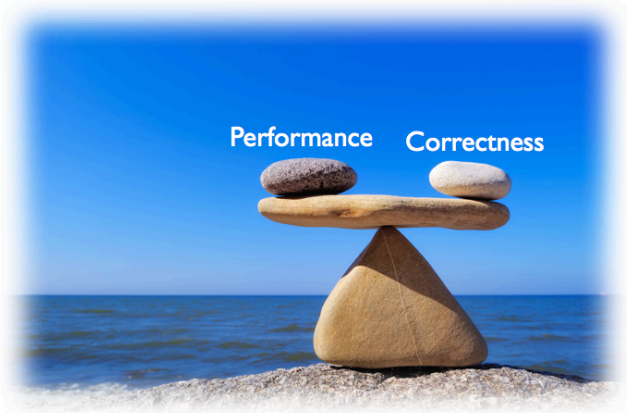
RQ3 Static Analysis

Class name	Concurrent Function Pairs ALL	DepCon	Reduction	Time (ms)
IntRange	351	21	16.71x	MIN 970
PerUserPoolDataSource	2,211	66	MAX 33.50x	1,535
SharedPoolDataSource	1,378	52	26.50x	1,046
DoubleIntIndex	595	297	MIN 2.00x	1,584
BufferedInputStream	55	22	2.50x	1,011
Vector	1,035	51	20.29x	1,733
Day	351	70	5.01x	1,448
NumberAxis	6,216	292	21.29x	1,156
PeriodAxis	8,001	278	28.78x	1,353
TimeSeries	861	296	2.91x	1,335
XYPlot	23,871	844	28.28x	2,252
XYSeries	325	114	2.85x	1,291
FileAppender	231	53	4.36x	971
WriterAppender	300	37	8.11x	1,064
XStream	2,211	427	5.18x	MAX 4,032
AVG	**3,199**	**195**	**13.89x**	**1,519**

- **Complete:** all bugs were always detected
- **(mostly) Precise:** **13.89x** of reduction (average)
- **Efficient:** **1,519 ms** on average
  (1.66% of the detection time)

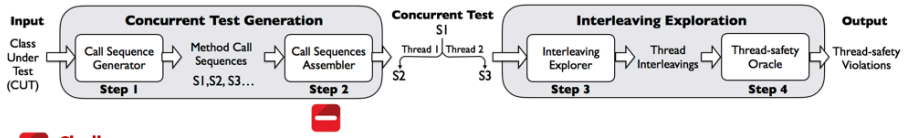
Conclusion

Synchronization is Challenging



6

Automated Concurrent Test Generation



Challenges

 # 1 – Step 2 Huge space of concurrent tests

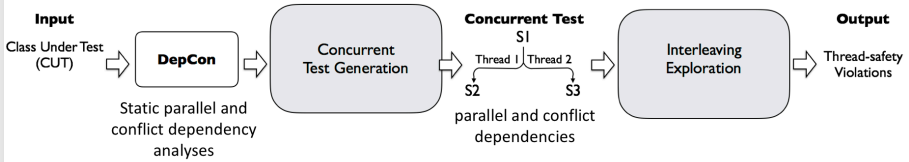
# of possible concurrent tests = M^{TL} **BufferedInpuStream** 10^{30}

M = # methods, L = max length of method call sequence, T = # threads (>= 3)

10 **10** **3**

17

DepCon



Cost-effective Static Analysis

java.io.BufferedInputStream JDK-4728096

Efficient: it takes 1 second

Effective: it reduces by 8.77x the number of generated tests
it exposes the bug 5.21x faster (on average)

30

RQ3 Static Analysis

Class name	Concurrent Function Pairs ALL	DepCon	Reduction	Time (ms)
IntRange	351	21	16.71x	MIN 970
PerUserPoolDataSource	2,211	66	MAX 33.50x	1,535
SharedPoolDataSource	1,378	52	26.50x	1,046
DoubleIntIndex	595	297	MIN 2.00x	1,584
BufferedInputStream	55	22	2.50x	1,011
Vector	1,035	51	20.29x	1,733
Day	351	70	5.01x	1,448
NumberAxis	6,216	292	21.29x	1,156
PeriodAxis	8,001	278	28.78x	1,353
TimeSeries	861	296	2.91x	1,335
XYPlot	23,871	844	28.28x	2,252
XYSeries	325	114	2.85x	1,291
FileAppender	231	53	4.36x	971
WriterAppender	300	37	8.11x	1,064
XStream	2,211	427	5.18x	MAX 4,032
AVG	**3,199**	**195**	**13.89x**	**1,519**

Complete: all bugs were always detected

(mostly) Precise: **13.89x** of reduction (average)

Efficient: **1,519 ms** on average

39

Thank you!

Questions?

Computing the Dependencies

```
public class A {
 private B field1 = new B();
 private int field2 = 0;

 public A() { … }

 public void m1() {
    B lock = field1;
    synchronized(lock){
     int k = m2();
    }
  }
}

private int m2() {
   field2++;
}
```

Method summaries

ACCESS SUMMARY : it represents an over-approximation of all the possible accesses of shared- memory locations performed by all possible invocations of under all possible execution paths.

{R(field1

LOCK SUMMARY : *set of locks that always protect every shared-memory accesses that can be triggered by an invocation of m:*

Concurrent Test for Thread-Safe Classes

Set of method call sequences that exercise
the public interface of a class from multiple threads

Concurrent Test

```
StringBufferInputStream var0 = new StringBufferInputStream("v;");
BufferedInputStream sout = new BufferedInputStream(var0);
```