Testing Concurrent and Distributed Systems

Mauro Pezzè
Università della Svizzera italiana (Lugano, Switzerland)
Università degli studi di Milano Bicocca (Milano, Italy)

Testing Concurrent and Distributed Systems

- ***concurrency and distribution***
  - fault types
  - testing framework
- ***classic approaches***
  - lockset
  - happens before
  - goodlock
- ***leading edge research***
  - relevant results
  - current trends and open problems
  - reproducing concurrent faults

Testing Concurrent and Distributed Systems

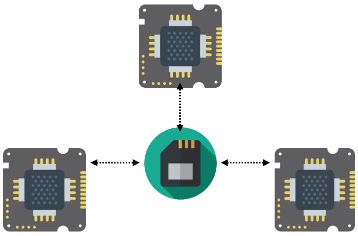
- ***concurrency and distribution***
  - ***fault types***
  - testing framework
- ***classic approaches***
  - lockset
  - happens before
  - goodlock
- ***leading edge research***
  - relevant results
  - current trends and open problems

---

concurrent and distributed systems

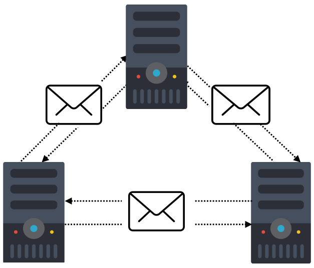
multiple execution flows that progress simultaneously

Multi-Threaded Systems



shared memory

Distributed Systems



message passing

serial execution

```
global balance = 0
```

Initially balance ≥ 0

```
def deposit(amt):
  b = balance + amt
  balance = b
  return balance
```

Preserves balance ≥ 0

```
def withdraw(amt):
  b = balance
  if b >= amt:
      balance = balance - amt
      return amt
  else:
      return 0
  fi
```

Preserves balance ≥ 0

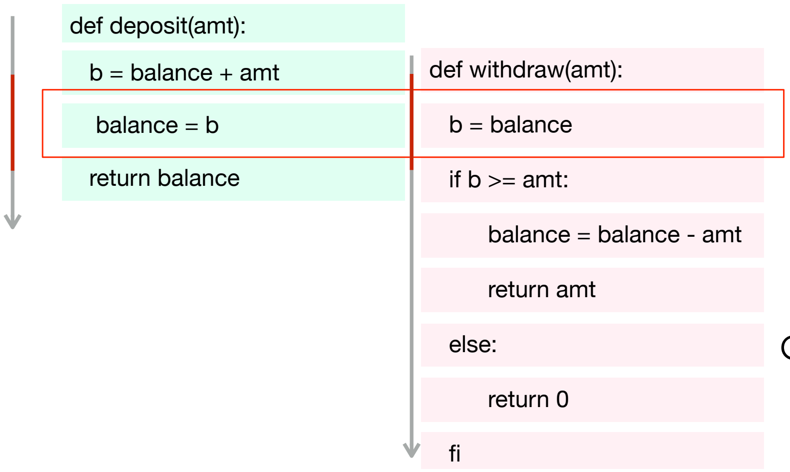
If executed serially
(one call at a time)
balance is always non-negative

serial execution preserves invariants

---

concurrent execution and race conditions

```
global balance = 0
```



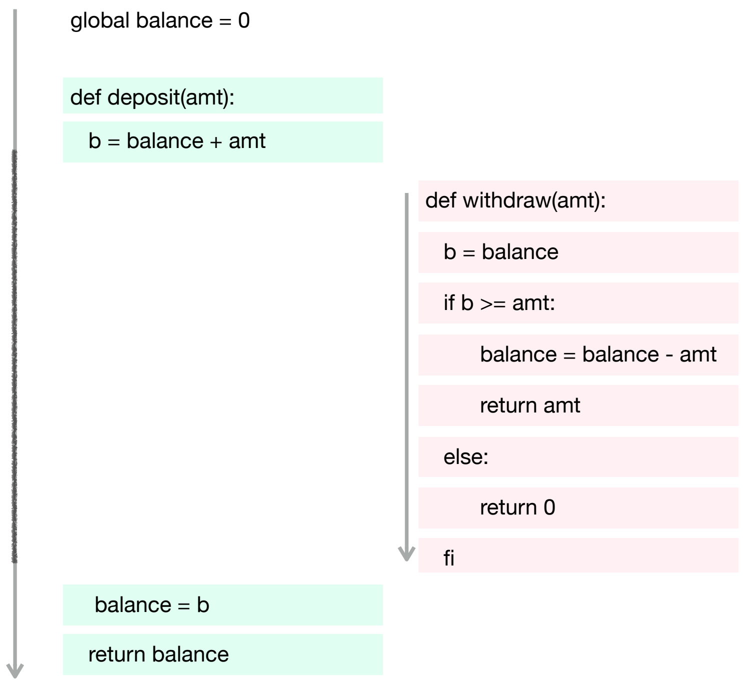
```
def deposit(amt):
  b = balance + amt
  balance = b
  return balance
```

```
def withdraw(amt):
  b = balance
  if b >= amt:
      balance = balance - amt
      return amt
  else:
      return 0
  fi
```

race condition
on balance

read-write race
deposit is writing balance
and withdraw is reading
balance

serializability



```
global balance = 0

def deposit(amt):
  b = balance + amt
```

```
def withdraw(amt):
  b = balance
  if b >= amt:
     balance = balance - amt
     return amt
  else:
     return 0
  fi
```

```
   balance = b
  return balance
```

deposit does not appear atomic
with respect to withdraw;
their executions are not serializable

serialisability violation

---

relaxed memory model

- Sequential consistency

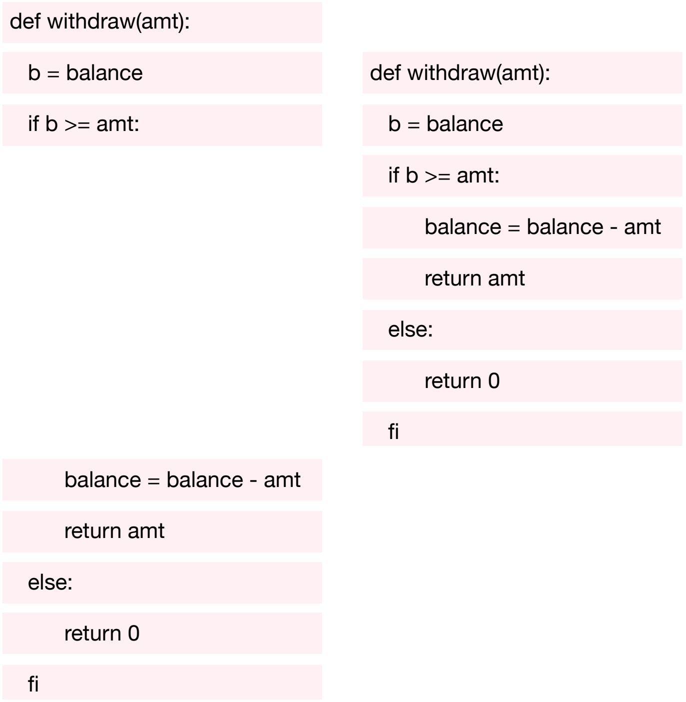
  standard memory model for reasoning about concurrent programs
- Modern hardware

  local write buffers, hierarchies of caches, speculative executions
  - significantly improve performance
  - invalidate SC in the presence of data races
- compilers' concurrency-oblivious optimizations

Relaxed memory models

formal sound semantics for realistic high-performance concurrency

different instances of the same object



```
def withdraw(amt):
    b = balance
    if b >= amt:
```

```
def withdraw(amt):
    b = balance
    if b >= amt:
        balance = balance - amt
        return amt
    else:
        return 0
    fi
```

```
        balance = balance - amt
        return amt
    else:
        return 0
    fi
```

race condition

atomicity violation

can occur
between two concurrent instances
of the same function or method

(suppose balance is 7 Krone,
and both withdrawals are for 5 Krone)

---

a data race …

```
1.  class Value {
2.    private int x = 1;
3.
4.    public synchronized void add(Value v){x = x + v.get();}
5.
6.    public int get(){return x;}
7.  }
8.
9.  class Task extends Thread{
10.   Value v1; Value v2;
11.
12.   public Task(Value v1,Value v2){
13.     this.v1 = v1; this.v2 = v2;
14.     this.start();
15.   }
16.
17.   public void run(){v1.add(v2);}
18. }
19.
20. class Main{
21.   public static void main(String[] args){
22.     Value v1 = new Value(); Value v2 = new Value();
23.     new Task(v1,v2); new Task(v2,v1);
24.   }
25. }
```

variable x:: class Value
unprotected access from the two Task threads (lines 4,6)

- one thread can call ***add method on object v1***, which ***calls the unsynchronized get method in the other object v2***.
- The other thread can make the ***dual operation***

*add method **synchronized***
does not prevent simultaneous application
*on **two different Value objects** by **two different threads***

Havelund: Using Runtime Analysis to Guide Model Checking of Java Programs

removed with a synchronized

Synchronized method get

```
class Value {
  private int x = 1;

  public synchronized void add(Value v){x = x + v.get();}

  public synchronized int get(){return x;}
}

class Task extends Thread{
  Value v1; Value v2;

  public Task(Value v1,Value v2){
    this.v1 = v1; this.v2 = v2;
    this.start();
  }

  public void run(){v1.add(v2);}
}

class Main{
  public static void main(String[] args){
    Value v1 = new Value(); Value v2 = new Value();
    new Task(v1,v2); new Task(v2,v1);
  }
}
```

---

leading to a deadlock

Synchronize method get

```
class Value {
  private int x = 1;

  public synchronized void add(Value v){x = x + v.get();}

  public synchronized int get(){return x;}
}

class Task extends Thread{
  Value v1; Value v2;

  public Task(Value v1,Value v2){
    this.v1 = v1; this.v2 = v2;
    this.start();
  }

  public void run(){v1.add(v2);}
}

class Main{
  public static void main(String[] args){
    Value v1 = new Value(); Value v2 = new Value();
    new Task(v1,v2); new Task(v2,v1);
  }
}
```

potential ***deadlock***:

- Task T1 locks V1
- Task T2 locks V2
- Task T1 waits for V2
- Task T2 waits for V1

limit concurrency to prevent data races

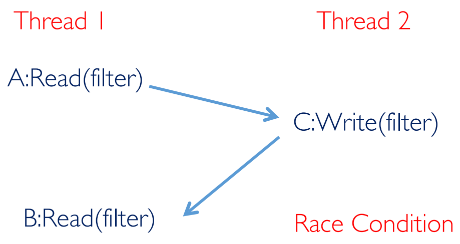
```
public class Logger{
…
private Filter filter;
//Thread 1
public void log(LogRecord record){
  if (record.getLevel().intValue() < levelValue
       || levelValue == offValue) {
    return;
  }
  synchronized (this) {                          Lock(this)
     if (filter != null){                        A:Read(filter)
        if( !filter.isLoggable(record)) {        B:Read(filter)
          return;
        }
     }.}                                         Unlock(this)
    …
}
// Thread 2
public void setFilter(Filter f) {
       this.filter = f;                          C:Write(filter)
}
```

locks (Java Synchronized)

---

but may fail

```
public class Logger{
…
private Filter filter;
//Thread 1
public void log(LogRecord record){
  if (record.getLevel().intValue() < levelValue
       || levelValue == offValue) {
    return;
  }
  synchronized (this) {                          Lock(this)
     if (filter != null){                        A:Read(filter)
        if( !filter.isLoggable(record)) {        B:Read(filter)
          return;
        }
     }.}                                         Unlock(this)
    …
}
// Thread 2
public void setFilter(Filter f) {
       this.filter = f;                          C:Write(filter)
}
```



Thread 1

Thread 2

A:Read(filter)

C:Write(filter)

B:Read(filter)

Race Condition

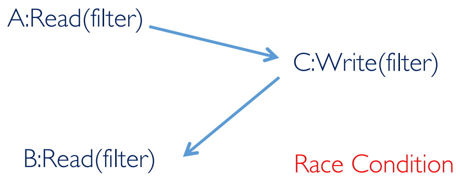
but may fail

```
public class Logger{
…
private Filter filter;
//Thread 1
public void log(LogRecord record){
  if (record.getLevel().intValue() < levelValue
       || levelValue == offValue) {
    return;
  }
  synchronized (this) {                           Lock(this)
     if (filter != null){                     A:Read(filter)
        if( !filter.isLoggable(record)) {     B:Read(filter)
            return;
        }
     }. }                                        Unlock(this)
    …
}
// Thread 2
public void setFilter(Filter f) {
       this.filter = f;                       C:Write(filter)
}
```

“'filter' is checked against null before being dereferenced.
This is done in a synchronized block to prevent 'filter'
from being set to null after it has been found to be non-null.

The problem is that setFilter() does not use synchronization at all,
and is explicitly allowed to set 'filter' to null.
The critical section in log(LogRecord) is thus completely useless.

Method setFilter() should be declared synchronized to avoid the race condition.
Method getFilter() should declared synchronized
otherwise the Java Memory Model allows it to return out-of-date values.”


A:Read(filter)
C:Write(filter)
B:Read(filter)
Race Condition

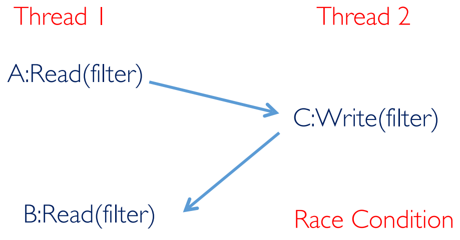
```
public class Logger{
…
private Filter filter;
//Thread 1
public void log(LogRecord record){
  if (record.getLevel().intValue() < levelValue
       || levelValue == offValue) {
    return;
  }
  synchronized (this) {                           Lock(this)
     if (filter != null){                     A:Read(filter)
        if( !filter.isLoggable(record)) {     B:Read(filter)
            return;
        }
     }. }                                        Unlock(this)
    …
}
// Thread 2
public void setFilter(Filter f) {
       this.filter = f;
}                                             C:Write(filter)
```


JDK / JDK-4779253

Race Condition in class java.util.logging.Logger

Details

Type:	Bug	Status:	CLOSED
Priority:	P4	Resolution:	Fixed
Affects Version/s:	1.4.0, 1.4.1, 7	Fix Version/s:	7
Component/s:	core-libs		
Labels:	noreg-trivial webbug		
Subcomponent:	java.util.logging		
Resolved In Build:	b16		
CPU:	generic, x86, sparc		
OS:	generic, solaris_7, windows_2000		
Verification:	Not verified		


Thread 1
Thread 2
A:Read(filter)
C:Write(filter)
B:Read(filter)
Race Condition

type of concurrency failures

data race

serializability/order violation

atomicity violation

deadlock

impact and frequency of concurrency failures

"… intermittently I get the following error"
[Apache, Bug #27315, Atomicity Violation]

"I've still no clues on why this crash occurs"
[MySQL, Bug #3596, Data Race]

"What should happen here, Charles?"
[Guava, Bug #976, Atomicity Violation]

hard to find

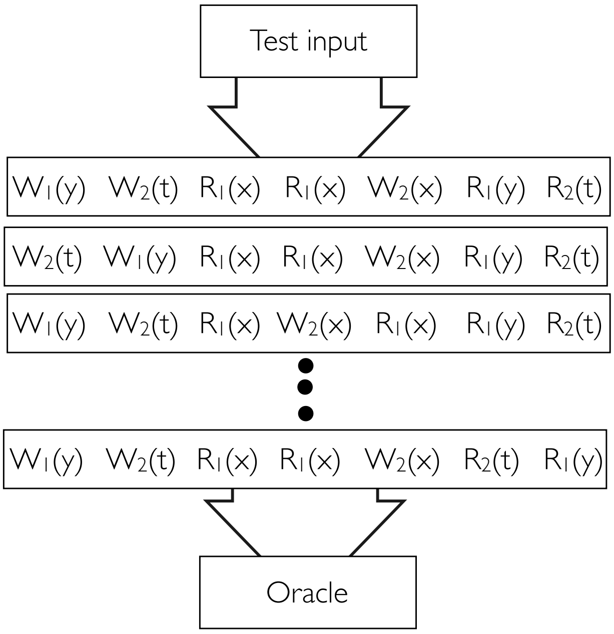
frequent

dangerous

Testing Concurrent and Distributed Systems

- ***concurrency and distribution***
  - fault types
  - ***testing framework***
- ***classic approaches***
  - lockset
  - happens before
  - goodlock
- ***leading edge research***
  - relevant results
  - current trends and open problems

Concurrent Test case



Test input

$W_1(y)$ $W_2(t)$ $R_1(x)$ $R_1(x)$ $W_2(x)$ $R_1(y)$ $R_2(t)$

$W_2(t)$ $W_1(y)$ $R_1(x)$ $R_1(x)$ $W_2(x)$ $R_1(y)$ $R_2(t)$

$W_1(y)$ $W_2(t)$ $R_1(x)$ $W_2(x)$ $R_1(x)$ $R_1(y)$ $R_2(t)$

⋮

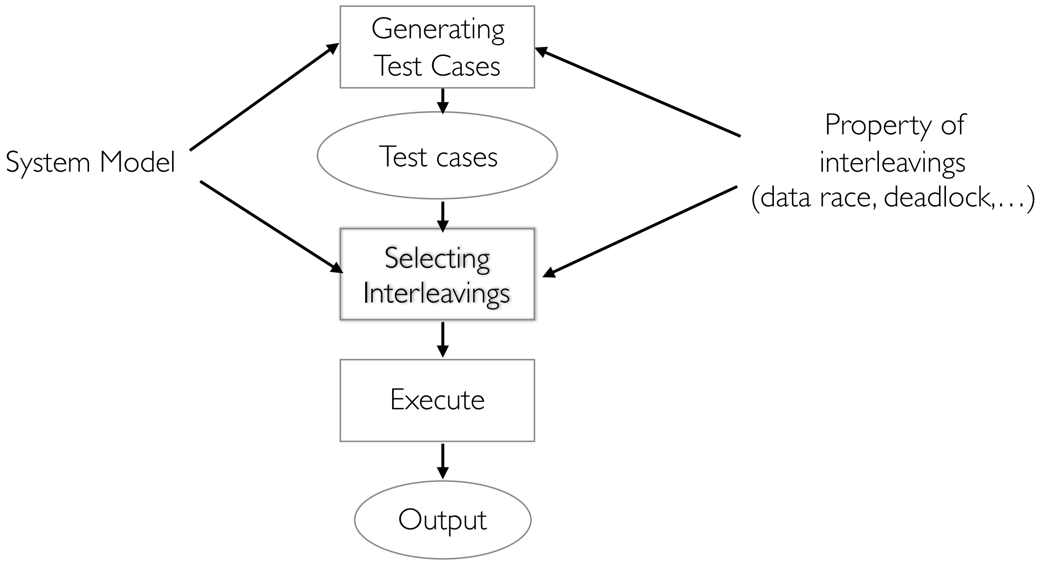
$W_1(y)$ $W_2(t)$ $R_1(x)$ $R_1(x)$ $W_2(x)$ $R_2(t)$ $R_1(y)$

Oracle

interleavings

<input, interleaving, oracle>

testing concurrent systems



Generating
Test Cases

System Model

Test cases

Property of
interleavings
(data race, deadlock,…)

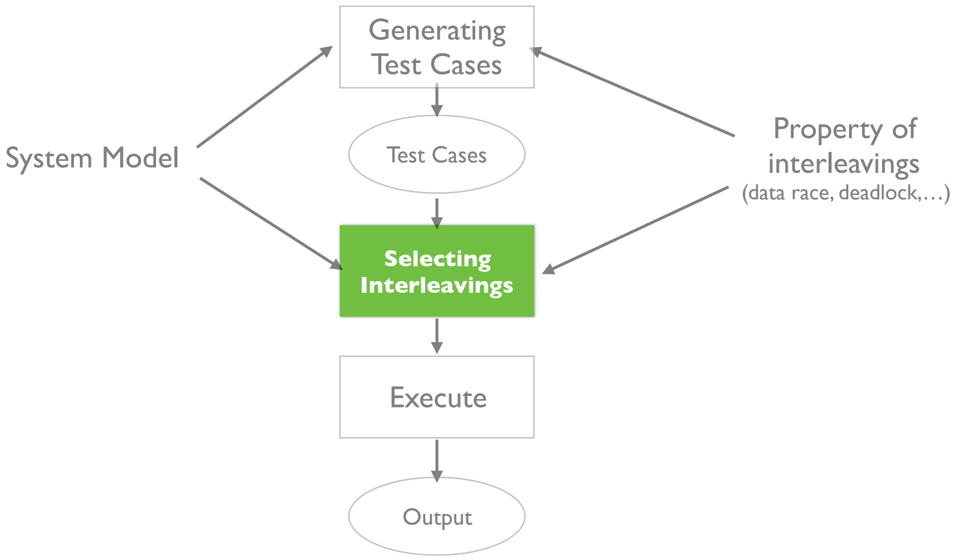
Selecting
Interleavings

Execute

Output

---

main focus of research



Generating
Test Cases

System Model

Test Cases

Property of
interleavings
(data race, deadlock,…)

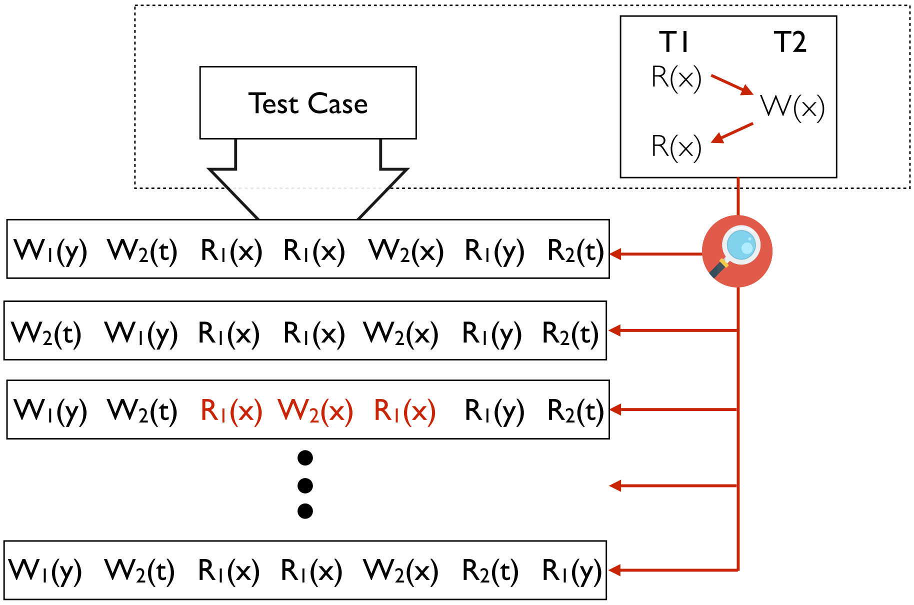
Selecting
Interleavings

Execute

Output

Property based

INPUT



Test Case

T1 T2

R(x)

W(x)

R(x)

$W_1(y)$ $W_2(t)$ $R_1(x)$ $R_1(x)$ $W_2(x)$ $R_1(y)$ $R_2(t)$

$W_2(t)$ $W_1(y)$ $R_1(x)$ $R_1(x)$ $W_2(x)$ $R_1(y)$ $R_2(t)$

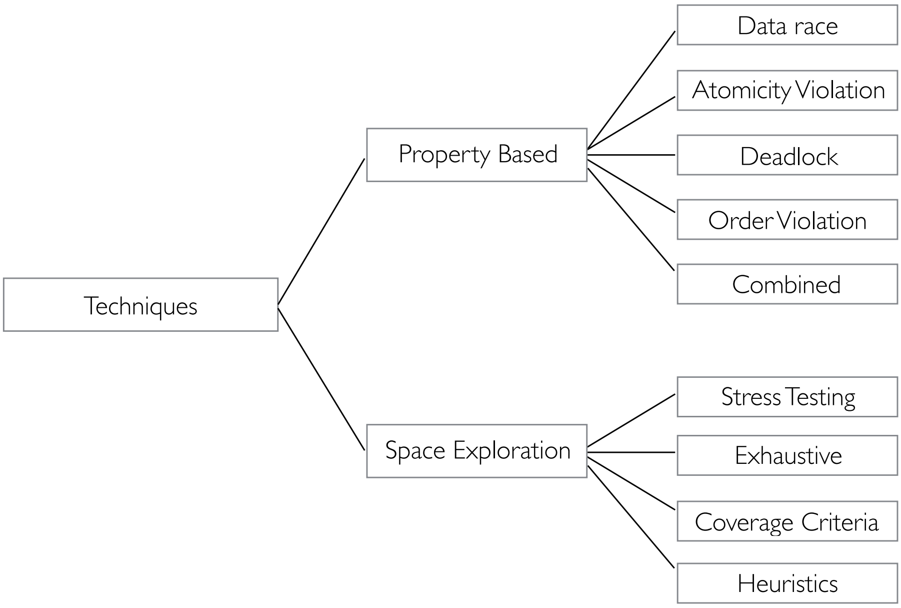
$W_1(y)$ $W_2(t)$ $R_1(x)$ $W_2(x)$ $R_1(x)$ $R_1(y)$ $R_2(t)$

⋮

$W_1(y)$ $W_2(t)$ $R_1(x)$ $R_1(x)$ $W_2(x)$ $R_2(t)$ $R_1(y)$

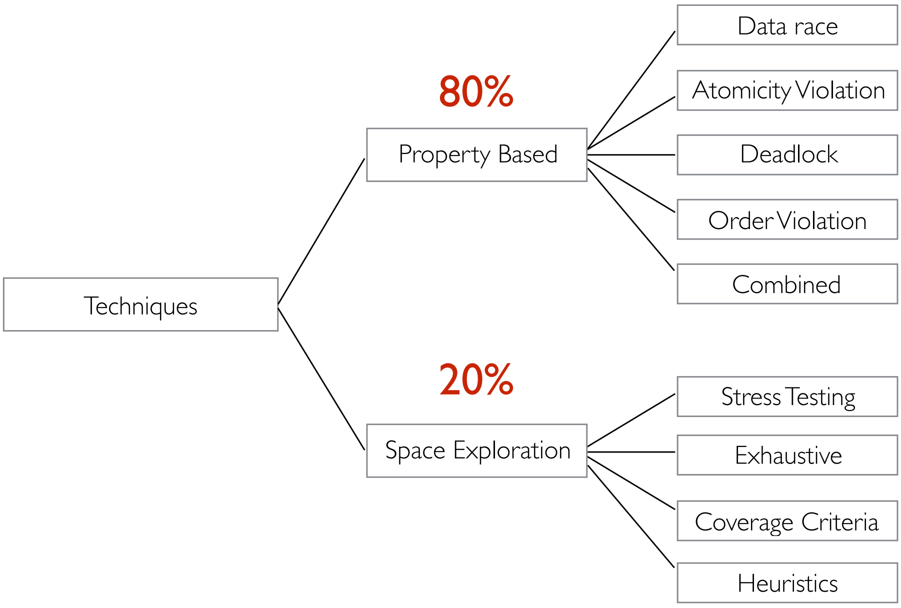
---

taxonomy



- Techniques
  - Property Based
    - Data race
    - Atomicity Violation
    - Deadlock
    - Order Violation
    - Combined
  - Space Exploration
    - Stress Testing
    - Exhaustive
    - Coverage Criteria
    - Heuristics

taxonomy



Techniques

80%

Property Based

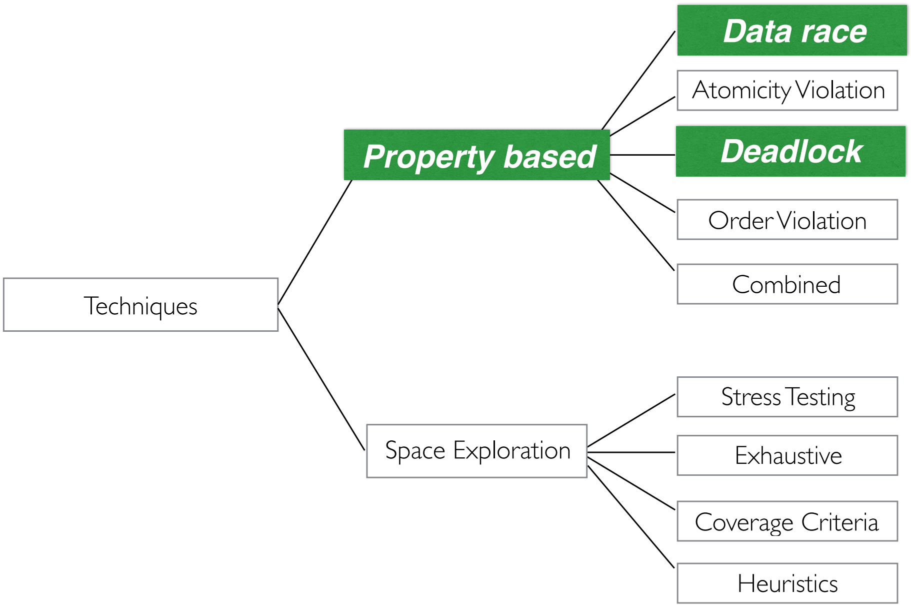
- Data race
- Atomicity Violation
- Deadlock
- Order Violation
- Combined

20%

Space Exploration

- Stress Testing
- Exhaustive
- Coverage Criteria
- Heuristics

seminal work



Techniques

Property based

- **Data race**
- Atomicity Violation
- **Deadlock**
- Order Violation
- Combined

Space Exploration

- Stress Testing
- Exhaustive
- Coverage Criteria
- Heuristics

lockset

happens before

goodlock

Testing Concurrent and Distributed Systems

- *concurrency and distribution*
  - fault types
  - *testing framework*
- ***classic approaches***
  - ***lockset***
  - happens before
  - goodlock
- *leading edge research*
  - relevant results
  - current trends and open problems

R. J. Lipton,
"Reduction: A method of proving properties of parallel programs,"
CACM 1975

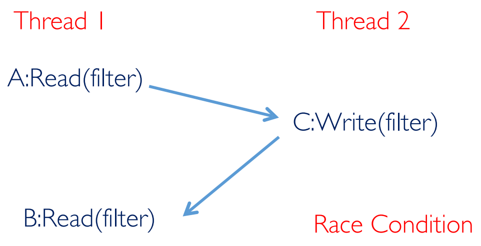
```
public class Logger{
…
private Filter filter;
//Thread 1
public void log(LogRecord record){
  if (record.getLevel().intValue() < levelValue
      || levelValue == offValue) {
    return;
  }
  synchronized (this) {                            Lock(this)
    if (filter != null){                           A:Read(filter)
      if( !filter.isLoggable(record)) {            B:Read(filter)
        return;
      }
    }. }                                           Unlock(this)
    …
}
// Thread 2
public void setFilter(Filter f) {
      this.filter = f;                             C:Write(filter)
}
```

The LockSet of an event is the set of locks held by a thread while executing the event.

LockSet(A) = {this}
LockSet(B) = {this}
LockSet(C) = ∅

LockSet Analysis: identifies shared memory accesses on different threads that are not protected by the same lock

Example: LockSet(A) ∩ LockSet(B) ∩ LockSet(C) = ∅



Dynamic Lockset Analysis

dynamically detecting ***violation of a locking discipline***

(set of rules to prevent data races)

Every variable shared between threads
must be protected by a mutual exclusion lock

Dynamic Lockset Analysis

INIT: each shared variable is associated with all available locks

RUN: thread accesses a shared variable:

intersect current set of candidate locks
with locks held by the thread

END: set of locks after executing a test

(set of locks always held by threads
accessing that variable)

empty set for v = no lock consistently protects v

Simple lockset analysis: example

Thread	*Program trace*	*Locks held*	*Lockset(x)*
		{}	{lck1, lck2}
thread A	*lock(lck1)*		
		{……………..}	
	x=x+1		
			{………………..}
	unlock(lck1}		
		{……………..}	
tread B	*lock{lck2}*		
		{……………..}	
	x=x+1		
			{………………..}
	unlock(lck2}		
		{……………..}	

Simple lockset analysis: example

Thread	*Program trace*	*Locks held*	*Lockset(x)*	
		{}	{lck1, lck2}	INIT: all locks for x
thread A	*lock(lck1)*			
		{lck1}		lck1 held
	x=x+1			
			{lck1}	Intersect with locks held
	unlock(lck1}			
		{}		
tread B	*lock{lck2}*			
		{lck2}		lck2 held
	x=x+1			
			{}	Empty intersection ***potential*** race
	unlock(lck2}			
		{}		

class logger

Java.util.loggin.Logger

```
public class Logger{
…
private Filter filter;
//Thread 1
public void log(LogRecord record){
 if (record.getLevel().intValue() <
levelValue
      || levelValue == offValue) {
   return;
 }                                          Lock(this)
 synchronized (this) {                      A:Read(filter)
   if (filter != null){                     B:Read(filter)
     if( !filter.isLoggable(record)) {
         return;
       }                                    Unlock(this)
   }. }
  …
}
// Thread 2
public void setFilter(Filter f) {           C:Write(filter)
      this.filter = f;
}
```

LockSet(A) = {…………………………}
LockSet(B) = {…..……………………}
LockSet(C) = {…………………………}

LockSet(A) ∩ LockSet(B) ∩ LockSet(C) = ………

---

class logger

Java.util.loggin.Logger

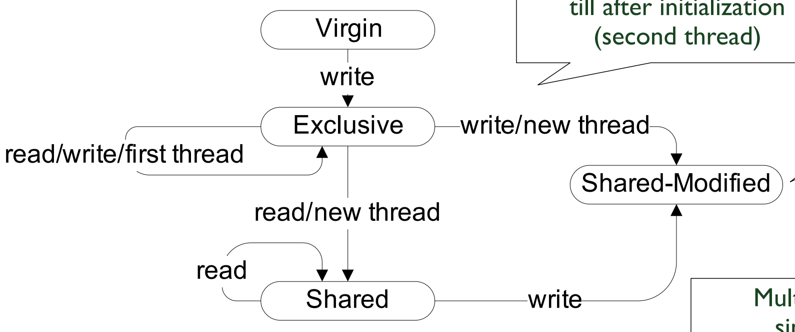
```
public class Logger{
…
private Filter filter;
//Thread 1
public void log(LogRecord record){
 if (record.getLevel().intValue() < levelValue
      || levelValue == offValue) {
   return;
 }
 synchronized (this) {                      Lock(this)
   if (filter != null){                     A:Read(filter)
     if( !filter.isLoggable(record)) {      B:Read(filter)
         return;
       }
   }. }                                     Unlock(this)
  …
}
// Thread 2
public void setFilter(Filter f) {
      this.filter = f;                      C:Write(filter)
}
```

LockSet(A) = {this}
LockSet(B) = {this}
LockSet(C) = ∅

LockSet(A) ∩ LockSet(B) ∩ LockSet(C) = ∅

Handling Realistic Cases

simple locking discipline violated by
- initialization of shared variables without holding a lock
- writing shared variables during initialization without locks
- allowing multiple readers in mutual exclusion with single writers



Delay analysis
till after initialization
(second thread)

Multiple writers
report violations

Multiple readers
single writer
do not report violations

Testing Concurrent and Distributed Systems

- ***concurrency and distribution***
  - ***fault types***
  - testing framework
- ***classic approaches***
  - lockset
  - ***happens before***
  - goodlock
- ***leading edge research***
  - relevant results
  - current trends and open problems

message passing
and
happens before

L. Lamport,
"Time, clocks, and the ordering of events in a distributed system,"
CACM 1978.

```
class Writer extends Actor {
 var results = ArrayBuffer[String]()
 def receive() = {
    case Write(result:String) =>
      results.append(result)
    case Flush => {
      writeToExternal(results)
      results = null
    }
  }
}

class Action(name:String, terminator:Terminator,
 writer:Writer) extends Actor {
  def receive() = {
     case Execute => {
       writer ! Write(name)
       terminator ! ActionDone
       }
   }
}

class Terminator(actionNum:Int, writer:Writer) extends Actor {
var curActions = actionNum
def receive() = {
     case ActionDone => {
        curActions -= 1
        if (curActions == 0) writer ! Flush
        }
    }
}
```

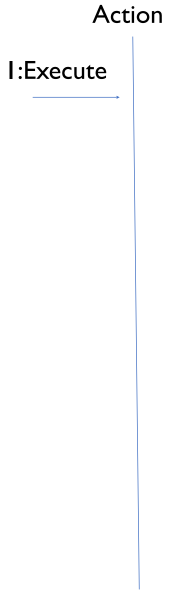
receive(Write)

receive(Flush)

send(Write)
send(ActionDone)

receive(ActionDone)

send(Flush)


Action

1:Execute

```
class Writer extends Actor {
 var results = ArrayBuffer[String]()
 def receive() = {
    case Write(result:String) =>
      results.append(result)
    case Flush => {
      writeToExternal(results)
      results = null
    }
  }
}

class Action(name:String, terminator:Terminator,
 writer:Writer) extends Actor {
  def receive() = {
     case Execute => {
       writer ! Write(name)
       terminator ! ActionDone
       }
   }
}

class Terminator(actionNum:Int, writer:Writer) extends Actor {
var curActions = actionNum
def receive() = {
     case ActionDone => {
        curActions -= 1
        if (curActions == 0) writer ! Flush
        }
    }
}
```

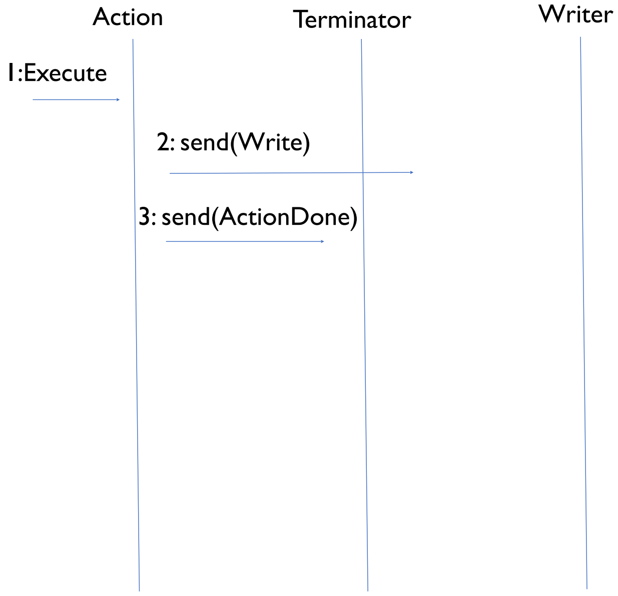
receive(Write)

receive(Flush)

send(Write)
send(ActionDone)

receive(ActionDone)

send(Flush)


Action

Terminator

Writer

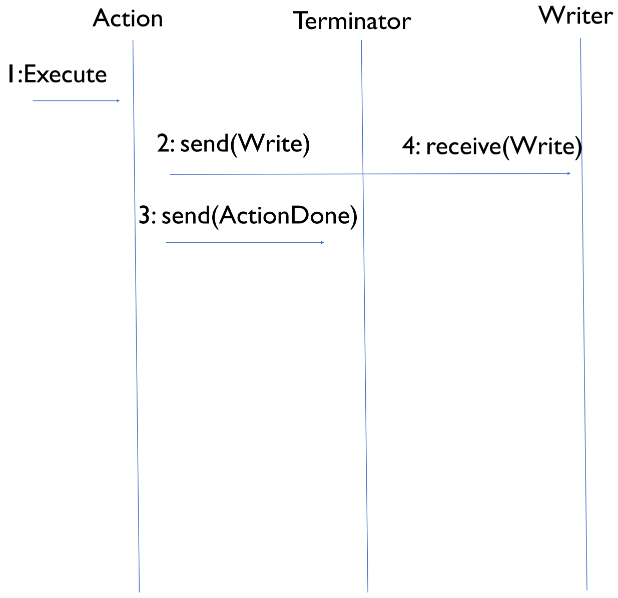
1:Execute

2: send(Write)

3: send(ActionDone)

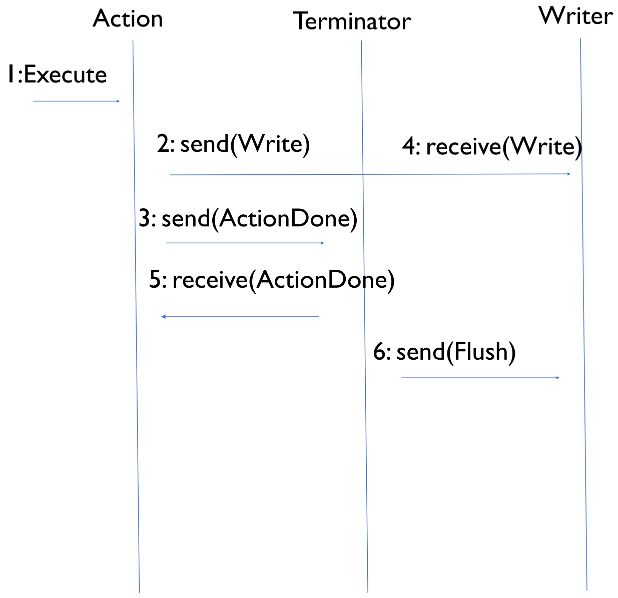
Msg Write is received by Writer and the append method is fine

```
class Writer extends Actor {
 var results = ArrayBuffer[String]()
 def receive() = {
    case Write(result:String) =>               receive(Write)
      results.append(result)
    case Flush => {                            receive(Flush)
      writeToExternal(results)
      results = null
    }
  }
}

class Action(name:String, terminator:Terminator,
 writer:Writer) extends Actor {
  def receive() = {
     case Execute => {
       writer ! Write(name)                    send(Write)
       terminator ! ActionDone           send(ActionDone)
       }
   }
}

class Terminator(actionNum:Int, writer:Writer) extends Actor {
var curActions = actionNum
def receive() = {
     case ActionDone => {              receive(ActionDone)
          curActions -= 1
          if (curActions == 0) writer ! Flush          send(Flush)
          }
    }
}
```



Action
Terminator
Writer
1:Execute
2: send(Write)
4: receive(Write)
3: send(ActionDone)

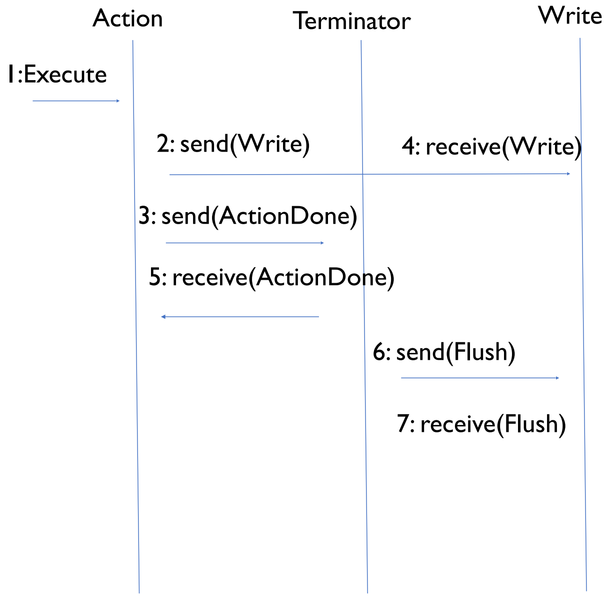
```
class Writer extends Actor {
 var results = ArrayBuffer[String]()
 def receive() = {
    case Write(result:String) =>               receive(Write)
      results.append(result)
    case Flush => {                            receive(Flush)
      writeToExternal(results)
      results = null
    }
  }
}

class Action(name:String, terminator:Terminator,
 writer:Writer) extends Actor {
  def receive() = {
     case Execute => {
       writer ! Write(name)                    send(Write)
       terminator ! ActionDone           send(ActionDone)
       }
   }
}

class Terminator(actionNum:Int, writer:Writer) extends Actor {
var curActions = actionNum
def receive() = {                      receive(ActionDone)
     case ActionDone => {
          curActions -= 1                      send(Flush)
          if (curActions == 0) writer ! Flush
          }
    }
}
```



Action
Terminator
Writer
1:Execute
2: send(Write)
4: receive(Write)
3: send(ActionDone)
5: receive(ActionDone)
6: send(Flush)

Msg Flush is received by Writer results is set to null

```
class Writer extends Actor {
 var results = ArrayBuffer[String]()
 def receive() = {
    case Write(result:String) =>                    receive(Write)
       results.append(result)
    case Flush => {                                 receive(Flush)
       writeToExternal(results)
       results = null
     }
   }
}

class Action(name:String, terminator:Terminator,
 writer:Writer) extends Actor {
   def receive() = {
       case Execute => {
          writer ! Write(name)                      send(Write)
          terminator ! ActionDone              send(ActionDone)
          }
     }
}

class Terminator(actionNum:Int, writer:Writer) extends Actor {
var curActions = actionNum
def receive() = {
       case ActionDone => {                   receive(ActionDone)
            curActions -= 1
            if (curActions == 0) writer ! Flush      send(Flush)
            }
     }
}
```



---

must happen before analysis

Given two events e_i and e_j

$e_i < e_j$ if:

- e_i and e_j belong to the same thread t and i < j
- e_i = send(msg_k) and e_j = receive(msg_k)

(a message is always sent before being received)

```
class Writer extends Actor {
 var results = ArrayBuffer[String]()
 def receive() = {
    case Write(result:String) =>                    receive(Write)
       results.append(result)
    case Flush => {                                  receive(Flush)
       writeToExternal(results)
       results = null
     }
   }
}

class Action(name:String, terminator:Terminator,
 writer:Writer) extends Actor {
  def receive() = {
     case Execute => {
       writer ! Write(name)                          send(Write)
       terminator ! ActionDone                       send(ActionDone)
       }
    }
}

class Terminator(actionNum:Int, writer:Writer) extends Actor {
var curActions = actionNum
def receive() = {
     case ActionDone => {                            receive(ActionDone)
         curActions -= 1                              send(Flush)
         if (curActions == 0) writer ! Flush
         }
     }
}
```

MUST HAPPENS BEFORE ANALYSIS

Given two events e_i and e_j

$e_i < e_j$ if:

e_i and e_j belong to the same thread t and i < j

e_i = send(msg_k) and e_j = receive(msg_k)

(a message is always sent before being received)

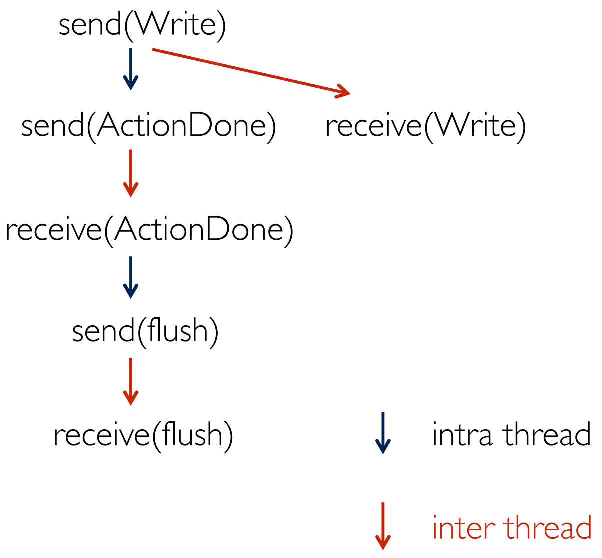
Happens-before relations:

-	send(Write) < send(ActionDone)	(intra thread)
-	send(Write) < receive(Write)	(inter thread)
-	send(ActionDone) < receive(ActionDone)	(inter thread)
-	receive(ActionDone) < send(Flush)	(intra thread)
-	send(Flush) < receive(Flush)	(inter thread)

Concurrent events:
- receive(ActionDone) and receive(Write)
- send(Flush) and receive(Write)
- receive(Write) and receive(Flush)

```
class Writer extends Actor {
 var results = ArrayBuffer[String]()
 def receive() = {
    case Write(result:String) =>                    receive(Write)
       results.append(result)
    case Flush => {                                  receive(Flush)
       writeToExternal(results)
       results = null
     }
   }
}

class Action(name:String, terminator:Terminator,
 writer:Writer) extends Actor {
  def receive() = {
     case Execute => {
       writer ! Write(name)                          send(Write)
       terminator ! ActionDone                  send(ActionDone)
       }
    }
}

class Terminator(actionNum:Int, writer:Writer) extends Actor {
var curActions = actionNum
def receive() = {
     case ActionDone => {                            receive(ActionDone)
         curActions -= 1                              send(Flush)
         if (curActions == 0) writer ! Flush
         }
     }
}
```

happens before relation


send(Write)

send(ActionDone)

receive(Write)

receive(ActionDone)

send(flush)

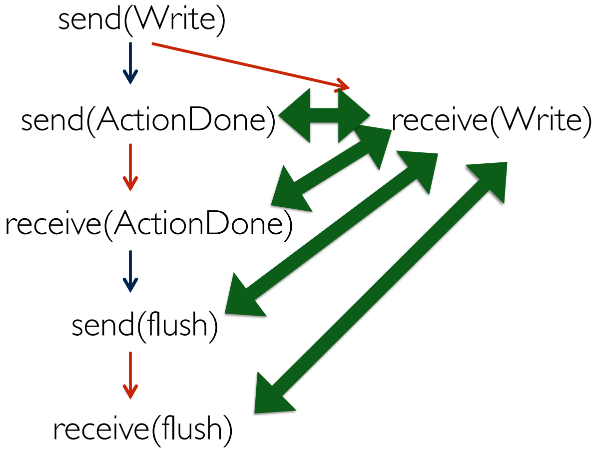
receive(flush)

intra thread

inter thread

concurrent events

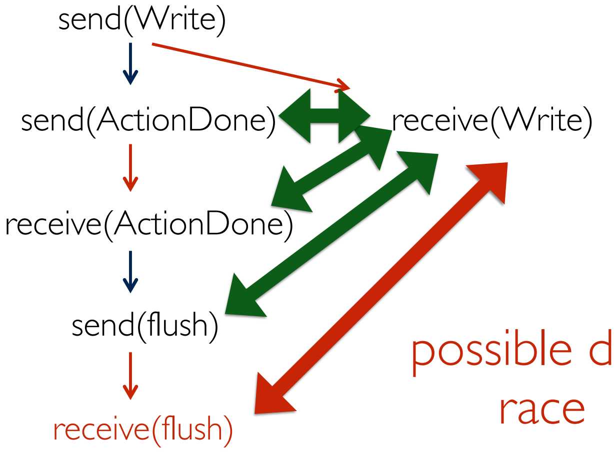
```
class Writer extends Actor {
 var results = ArrayBuffer[String]()
 def receive() = {
    case Write(result:String) =>                receive(Write)
       results.append(result)
    case Flush => {                              receive(Flush)
       writeToExternal(results)
       results = null
     }
  }
}

class Action(name:String, terminator:Terminator,
 writer:Writer) extends Actor {
   def receive() = {
       case Execute => {
          writer ! Write(name)                      send(Write)
          terminator ! ActionDone             send(ActionDone)
          }
    }
}

class Terminator(actionNum:Int, writer:Writer) extends Actor {
var curActions = actionNum
def receive() = {
       case ActionDone => {               receive(ActionDone)
          curActions -= 1
          if (curActions == 0) writer ! Flush           send(Flush)
          }
     }
}
```



---

concurrent events

```
class Writer extends Actor {
 var results = ArrayBuffer[String]()
 def receive() = {
    case Write(result:String) =>                receive(Write)
       results.append(result)
    case Flush => {                              receive(Flush)
       writeToExternal(results)
       results = null
     }
  }
}

class Action(name:String, terminator:Terminator,
 writer:Writer) extends Actor {
   def receive() = {
       case Execute => {
          writer ! Write(name)                      send(Write)
          terminator ! ActionDone             send(ActionDone)
          }
    }
}

class Terminator(actionNum:Int, writer:Writer) extends Actor {
var curActions = actionNum
def receive() = {
       case ActionDone => {               receive(ActionDone)
          curActions -= 1
          if (curActions == 0) writer ! Flush           send(Flush)
          }
     }
}
```



possible data race

```
class Writer extends Actor {
 var results = ArrayBuffer[String]()
 def receive() = {
    case Write(result:String) =>                    receive(Write)
       results.append(result)
    case Flush => {                                  receive(Flush)
       writeToExternal(results)
       results = null
     }
   }
}

class Action(name:String, terminator:Terminator,
 writer:Writer) extends Actor {
   def receive() = {
       case Execute => {
          writer ! Write(name)                         send(Write)
          terminator ! ActionDone                 send(ActionDone)
          }
     }
}

class Terminator(actionNum:Int, writer:Writer) extends Actor {
var curActions = actionNum
def receive() = {
       case ActionDone => {                   receive(ActionDone)
            curActions -= 1
            if (curActions == 0) writer ! Flush              send(Flush)
            }
     }
}
```

MUST HAPPENS BEFORE ANALYSIS:

Given two events e_i and e_j, e_i < e_j if:

- e_i and e_j belong to the same thread *t* and *i* < *j*
- e_i = send(msg_k) and e_j = receive(msg_k) (a message is always sent before being received)

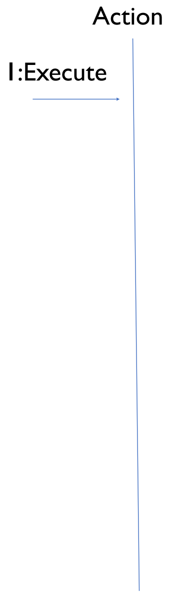
Happens-before relations:

- send(Write) > send(ActionDone)	(intra thread)
- send(Write) > receive(Write)	(inter thread)
- send(ActionDone) > receive(ActionDone)	(inter thread)
- receive(ActionDone) > send(Flush)	(intra thread)
- send(Flush) > receive(Flush)	(inter thread)

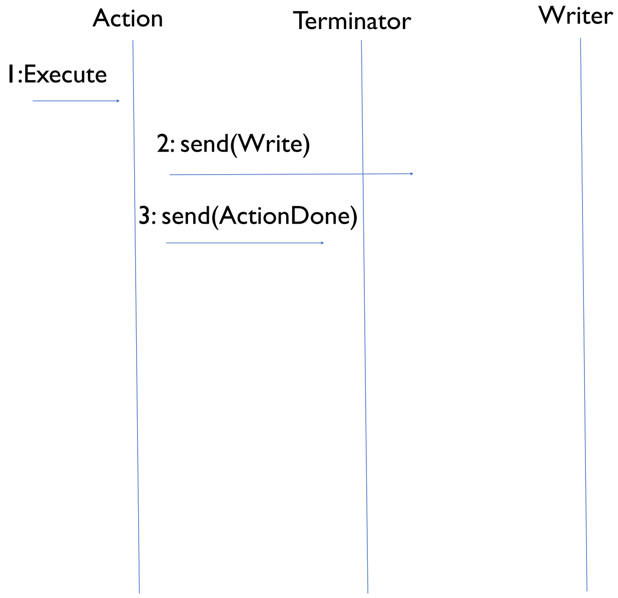
Concurrent events:

- receive(ActionDone) and receive(Write)
- send(Flush) and receive(Write)
- receive(Write) and receive(Flush)

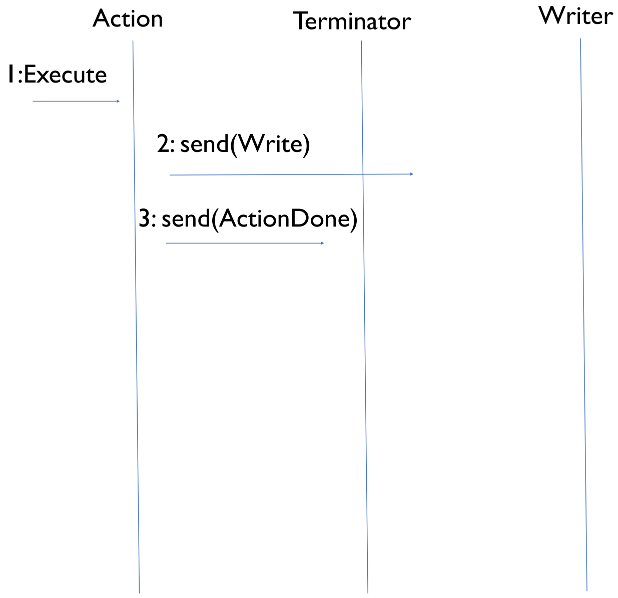
```
class Writer extends Actor {
 var results = ArrayBuffer[String]()
 def receive() = {
    case Write(result:String) =>                    receive(Write)
       results.append(result)
    case Flush => {                                  receive(Flush)
       writeToExternal(results)
       results = null
     }
   }
}

class Action(name:String, terminator:Terminator,
 writer:Writer) extends Actor {
   def receive() = {
       case Execute => {
          writer ! Write(name)                         send(Write)
          terminator ! ActionDone                 send(ActionDone)
          }
     }
}

class Terminator(actionNum:Int, writer:Writer) extends Actor {
var curActions = actionNum
def receive() = {
       case ActionDone => {                   receive(ActionDone)
            curActions -= 1
            if (curActions == 0) writer ! Flush              send(Flush)
            }
     }
}
```



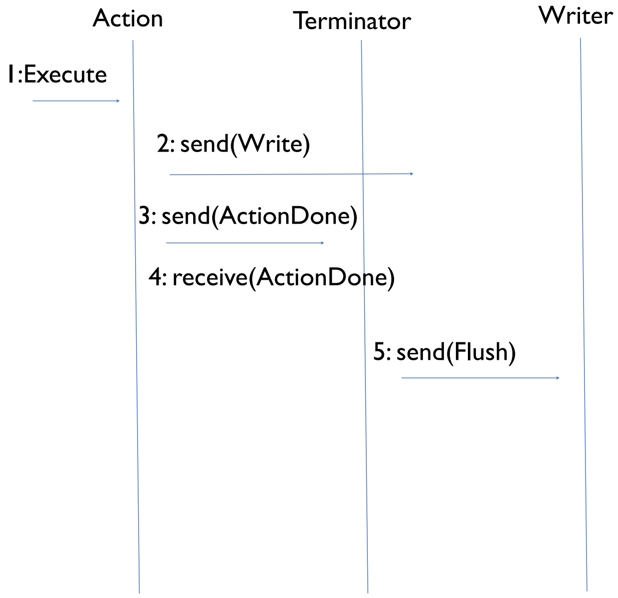
```
class Writer extends Actor {
 var results = ArrayBuffer[String]()
 def receive() = {
    case Write(result:String) =>                  receive(Write)
       results.append(result)
    case Flush => {                               receive(Flush)
       writeToExternal(results)
       results = null
     }
   }
}

class Action(name:String, terminator:Terminator,
 writer:Writer) extends Actor {
  def receive() = {
      case Execute => {
         writer ! Write(name)                      send(Write)
         terminator ! ActionDone               send(ActionDone)
         }
    }
}

class Terminator(actionNum:Int, writer:Writer) extends Actor {
var curActions = actionNum
def receive() = {
      case ActionDone => {                receive(ActionDone)
          curActions -= 1
          if (curActions == 0) writer ! Flush          send(Flush)
          }
     }
}
```



```
class Writer extends Actor {
 var results = ArrayBuffer[String]()
 def receive() = {
    case Write(result:String) =>                  receive(Write)
       results.append(result)
    case Flush => {                               receive(Flush)
       writeToExternal(results)
       results = null
     }
   }
}

class Action(name:String, terminator:Terminator,
 writer:Writer) extends Actor {
  def receive() = {
      case Execute => {
         writer ! Write(name)                      send(Write)
         terminator ! ActionDone               send(ActionDone)
         }
    }
}

class Terminator(actionNum:Int, writer:Writer) extends Actor {
var curActions = actionNum
def receive() = {
      case ActionDone => {                receive(ActionDone)
          curActions -= 1
          if (curActions == 0) writer ! Flush          send(Flush)
          }
     }
}
```



```
class Writer extends Actor {
 var results = ArrayBuffer[String]()
 def receive() = {
    case Write(result:String) =>                receive(Write)
       results.append(result)
    case Flush => {                             receive(Flush)
       writeToExternal(results)
       results = null
     }
  }
}

class Action(name:String, terminator:Terminator,
 writer:Writer) extends Actor {
  def receive() = {
      case Execute => {
         writer ! Write(name)                       send(Write)
         terminator ! ActionDone              send(ActionDone)
         }
    }
}

class Terminator(actionNum:Int, writer:Writer) extends Actor {
var curActions = actionNum
def receive() = {
      case ActionDone => {              receive(ActionDone)
           curActions –= 1
           if (curActions == 0) writer ! Flush         send(Flush)
           }
     }
}
```



Action

Terminator

Writer

1:Execute

2: send(Write)

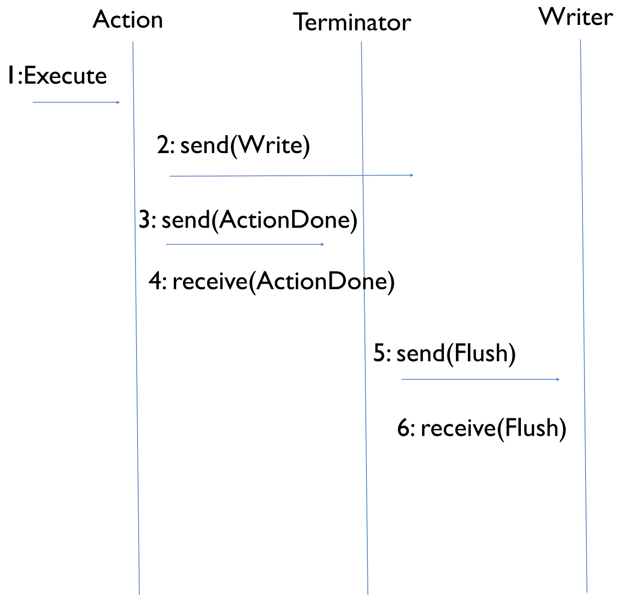
3: send(ActionDone)

4: receive(ActionDone)

5: send(Flush)

Msg Flush is received by Writer and results is set to null

```
class Writer extends Actor {
 var results = ArrayBuffer[String]()
 def receive() = {
    case Write(result:String) =>                receive(Write)
       results.append(result)
    case Flush => {                             receive(Flush)
       writeToExternal(results)
       results = null
     }
  }
}

class Action(name:String, terminator:Terminator,
 writer:Writer) extends Actor {
  def receive() = {
      case Execute => {
         writer ! Write(name)                       send(Write)
         terminator ! ActionDone              send(ActionDone)
         }
    }
}

class Terminator(actionNum:Int, writer:Writer) extends Actor {
var curActions = actionNum
def receive() = {
      case ActionDone => {              receive(ActionDone)
           curActions –= 1
           if (curActions == 0) writer ! Flush         send(Flush)
           }
     }
}
```



Action

Terminator

Writer

1:Execute

2: send(Write)

3: send(ActionDone)

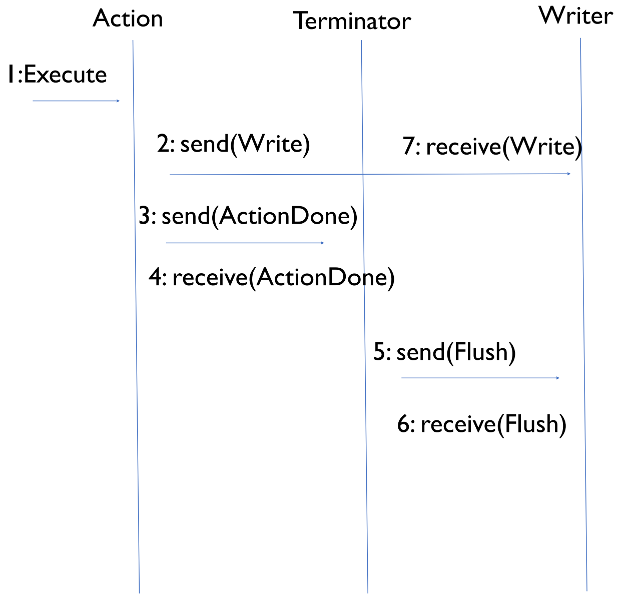
4: receive(ActionDone)

5: send(Flush)

6: receive(Flush)

Msg Write is received by Writer and a null pointer exception is thrown

```
class Writer extends Actor {
 var results = ArrayBuffer[String]()
 def receive() = {
    case Write(result:String) =>               receive(Write)
        results.append(result)
    case Flush => {                            receive(Flush)
        writeToExternal(results)
        results = null
      }
   }
}

class Action(name:String, terminator:Terminator,
 writer:Writer) extends Actor {
   def receive() = {
        case Execute => {
           writer ! Write(name)                 send(Write)
           terminator ! ActionDone         send(ActionDone)
           }
      }
}

class Terminator(actionNum:Int, writer:Writer) extends Actor {
var curActions = actionNum
def receive() = {
        case ActionDone => {          receive(ActionDone)
             curActions -= 1
             if (curActions == 0) writer ! Flush      send(Flush)
             }
        }
}
```



Testing Concurrent and Distributed Systems

- *concurrency and distribution*
  - fault types
  - testing framework
- ***classic approaches***
  - lockset
  - happens before
  - ***goodlock***
- *leading edge research*
  - relevant results
  - current trends and open problems

```
class Value {
  private int x = 1;

  public synchronized void add(Value v){x = x + v.get();}

  public synchronized int get(){return x;}
}

class Task extends Thread{
  Value v1; Value v2;

  public Task(Value v1,Value v2){
    this.v1 = v1; this.v2 = v2;
    this.start();
  }

  public void run(){v1.add(v2);}
}

class Main{
  public static void main(String[] args){
    Value v1 = new Value(); Value v2 = new Value();
    new Task(v1,v2); new Task(v2,v1);
  }
}
```

potential deadlock:

- Task T1 locks V1
- Task T2 locks V2
- Task T1 waits for V2
- Task T2 waits for V1

Havelund: Using Runtime Analysis to Guide Model Checking of Java Programs

---

Goodlock algorithm

AT RUNTIME:

record the locking pattern for each thread during runtime as a lock tree

one lock tree per tread == nested pattern in which locks are taken by the thread

AFTER EXECUTION:

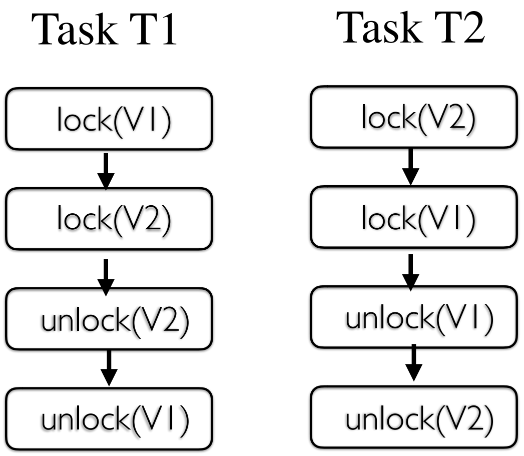
compare the trees for each pair of threads

for each pair of trees $<t_1, t_2>$ and each operation on a shared memory location n_1 of t_1

check that no lock below n_1 in t_1 is above a node n_2 in a thread t_2

deadlock

```
class Value {
  private int x = 1;

  public synchronized void add(Value v){x = x + v.get();}

  public synchronized int get(){return x;}
}

class Task extends Thread{
  Value v1; Value v2;

  public Task(Value v1,Value v2){
    this.v1 = v1; this.v2 = v2;
    this.start();
  }

  public void run(){v1.add(v2);}
}

class Main{
  public static void main(String[] args){
    Value v1 = new Value(); Value v2 = new Value();
    new Task(v1,v2); new Task(v2,v1);
  }
}
```



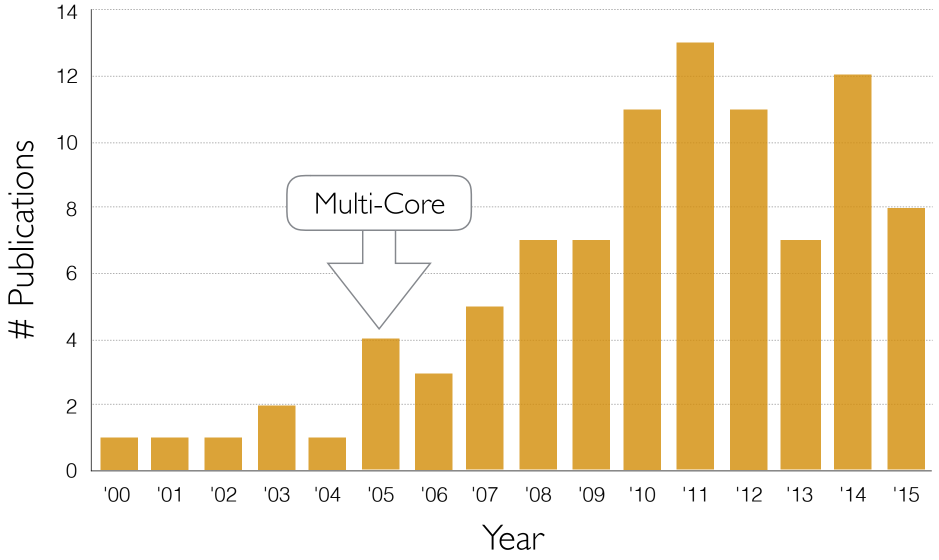
Havelund: Using Runtime Analysis to Guide Model Checking of Java Programs

---

Testing Concurrent and Distributed Systems

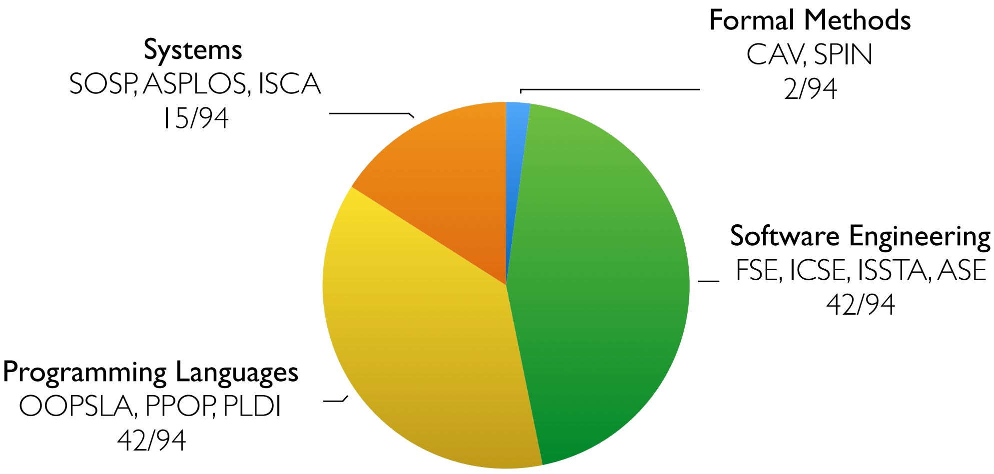
- ***concurrency and distribution***
  - ***fault types***
  - testing framework
- ***classic approaches***
  - lockset
  - happens before
  - goodlock
- ***leading edge research***
  - relevant results
  - current trends and open problems

Research landscape

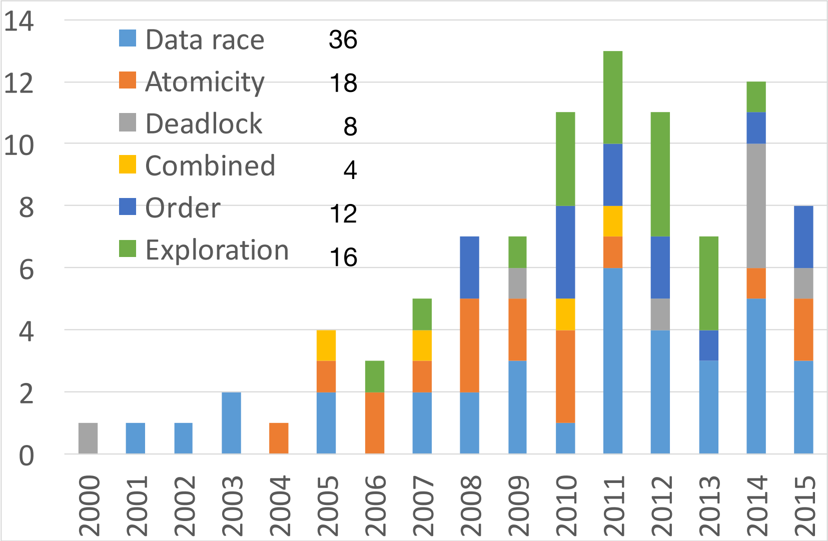


90+ Techniques presented from 2000 in top-tier venues

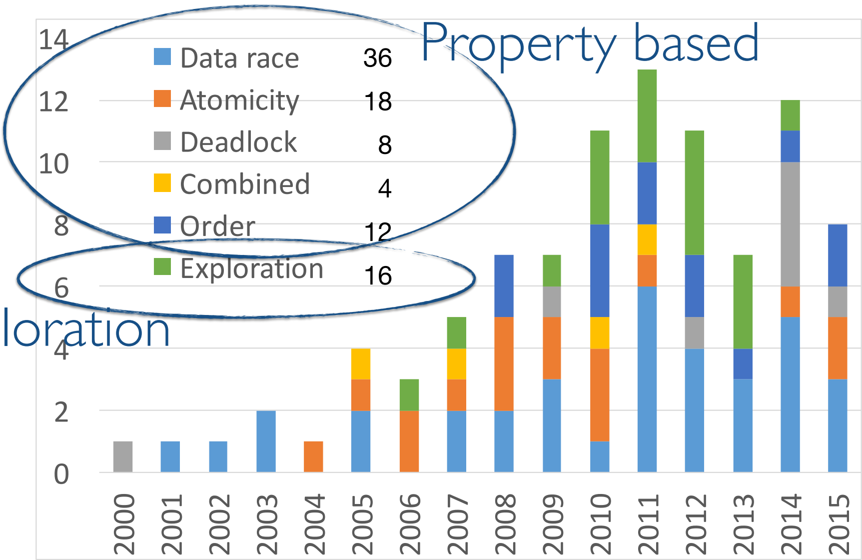
Research communities



Research focus



Research focus



Property based

space exploration

Property based

improving precision of ***happens-before*** analysis

to detect ***data-races*** and ***atomicity*** violations

improving ***performance*** of ***happens-before*** and ***good-lock*** analyses

to detect ***data-races*** and ***deadlocks***

improving ***recall*** of ***happens-before*** analysis to detect ***data-races***

extending ***happens-before*** analysis to ***Web***, ***event-based*** and ***Android***

extending ***happens-before*** analysis to ***relaxed memory models*** (C++, Java)

complementing with ***test case generation***

to detect ***data-races, atomicity violations*** and ***deadlocks***

violations of ***correctness properties***

Improving precision of happens-before analysis to detect atomicity violations and data races

Atomicity violations	data races
[Velodrome PLDI'08]	[RaceFuzzer PLDI'08]
[AtomFuzzer FSE'08]	[Frost SOSP'11]
[Penelope FSE'10]	[Portend ASPLOS'12]

improving precision of happens-before analysis to detect atomicity violations

'08 Velodrome' cyclic patterns

- reduces ***false positives*** by looking for ***cyclic patterns*** in the happens-before graph
  (sufficient and necessary conditions for atomicity violations)

'08 AtomFuzzer's atomic specification

- exploits annotations that ***specify*** which code ***blocks*** are intended to be atomic
- limits the analysis to pairs of execution flows that use a ***single lock*** to ensure the atomicity of a code region
- randomly generates interleavings by exploiting ***happens-before*** analysis to capture order relations among flows
- executes the test case with ***random pauses*** in correspondence of accesses to critical memory regions
  to maximize the probability of observing an atomicity violation

'10 Penelope's atomicity violation patterns

- considers alternative orders of lock acquisitions and releases that violate predefined ***atomicity violation patterns***
- re-executes the target program under the predicted schedules to ***prune false positives*** with oracles

---

improving precision of happens-before analysis to detect data races

'08 Frost

- detects ***non-benign data races*** by ***comparing results*** and program state of multiple replicas
  of the same program with different interleavings
- ***segments*** an execution into ***epochs***, and runs each epoch on three replicas
  - executes a replica with dynamic ***happens-before*** analysis to detect synchronization points in the program
  - executes the other two replicas with a non-preemptive controlled scheduler on a single thread

'11 RaceFuzzer's order information

- dynamically ***computes order information*** using an imprecise but efficient
  combines ***lockset*** and ***happens-before*** analyses to reduce computational cost

'12 Portend's classification of data races

- precisely ***classifies data races***, based on the effects on the system under test
  - considers data races as ***benign*** if they produce same results state with all tests
- checks the property with ***symbolic execution***

Improving performance of happens-before and deadlock analysis to detect data races, atomicity violations and deadlocks

data races	atomicity violations	deadlocks
[FastTrack PLDI'09]	[Falcon ICSE'10]	[MagicFuzzer ICSE'12]
[LiteRace PLDI'09]		[ConLock ICSE'14]
[Carisma ISSTA'12]		

---

improving performance of happens-before to detect data races

'09 Fastrack's lightweight representation

- proposes a ***lightweight representation of the happens-before information*** that records only the information about the last write operation on each data item
- reduces the cost of vector clock comparison up to an order of magnitude

'09 LiteRace's cold regions

- introduces ***sampling*** to reduce analysis overhead
- ***instruments only cold regions*** defined as the less frequently accessed code elements
- assumption: frequently accessed code elements (hot regions) less likely to be involved in data races

'12 Carisma's similarity relation

- exploits ***similarity between multiple accesses*** to the same data structures,
- ***dynamically infers the application contexts*** and uses the contexts to compute the distribution of memory locations across data structure to better balance the sampling budget

improving performance of happens-before analysis to detect atomicity violations

’10 Falcon’s siding window

- refers to ***fixed-sized sliding window*** to detect suspicious ***patterns*** that lead to unserializable memory accesses
  - maintains access information for each shared data item in a fixed-size window,
  - uses the information stored in a window to detect suspicious memory access patterns
- The sliding window keeps ***focus on the closely related accesses***

---

improving performance (scalability) to detect deadlocks

’12 MagicFuzzer’s detectors of cycles in the lock graph

- prunes the ***good lock*** graph:

  a ***deadlock*** that corresponds to a ***cycle in the lock graph*** contains only nodes that have both incoming and outgoing edges
- iteratively removes all the nodes that do not satisfy this property
- uses a novel algorithm to ***analyse the pruned graph***
- ***partitions the nodes*** based on the execution flows, and does not explore redundant paths

’12 ConLock’s should-happen before relation

- addresses the ***thrashing problem*** of randomized scheduling algorithms::

  randomized scheduler generates artificial deadlocks: the execution flows are suspended by the scheduler and cannot progress, but a deadlock cannot be confirmed.
- introduces a ***should-happen-before order relation*** computed with dynamic analysis to increase the probability to reach and thus confirm a deadlock

Improving recall of happens-before analysis to detect data races

[Smaragdakis et al. POPL'12]

[RVPredict PLDI'14]

[DrFinder FSE'15]

Improving recall of happens-before analysis to detect data races (i/ii)

'12 Smaragdakis et al.'s causally precedes relation

- PROBLEM: ***happens-before*** analysis focus on single execution traces
  thus may infer incorrect order relations and miss some data races
- introduce ***causally-precedes*** analysis to mitigate the problem:
  based on a ***new causally-precedes*** (CP) relation that relaxes the happens-before relation with respect to lock releases and acquisitions detect CP-races that occur when two conflicting memory accesses are not CP related

'14 RVPredict's order relation

- defines an ***order relation*** to detect data races that improves the accuracy of CP-analysis
- takes into account ***control flow information***

Improving recall of happens-before analysis to detect data races (ii/ii)

'15 DrFinder' may trigger relation

- PROBLEM: ***hidden data race***
  == pair of accesses to the same shared memory location in a happens-before relation only for some interleavings
  - not revealed with happens-before and extensions due to the over-constraining nature of the analysis
- INTUITION: many hidden races can be detected by ***reversing the order of execution*** of one or more operations in a happens-before relation
  - computes ***may-trigger relation*** on an execution trace
  - looks for alternative interleavings that might expose data races,
  - executes the selected interleavings to check their feasibility.

Extending happens-before to new paradigms to detect data races

Web	Event-based	Android
[WebRacer PLDI'12]	[EventRacer OOPSLA'13]	[DroidRacer PLDI'14]

new paradigms

’12 WebRacer

- **happens-before** analysis enhanced with the semantics of **Web** platforms — focus on
  - variable races == **data races** caused by **concurrent accesses to shared memory locations**
  - HTML races == **accesses of DOM nodes** may occur both **before** and **after creations**
  - function races == **function invocations** occur both **before** and **after parsing** the functions
  - event dispatch races == **events** fire both **before** and **after adding the event handlers**

’13 EventRacer

- happens-before analysis **for event-based programs**

’14 DroidRacer

- exploits concurrency semantics of **Android** programming model to derive precise **happens-before** relation to reduces of false positives

---

happens-before for relaxed memory models

C++	Java	Android
[MultiRace PPoPP’03]	[Java RaceFinder ASE’09]	[Relaxer ISSTA’11]

relaxed memory models

’03 MultiRace

- combines **lockset** and **happens-before** analyses
- takes into account both **lock-based** and **barrier synchronization** mechanisms
- detects **data races** in production mode

’09 Java RaceFinder

- introduces new **happens-before** analysis to capture ordering relations in the **relaxed Java memory model**
- relies on **Java PathFinder** to generate interleavings that may result in data races
- explores the interleaving space driven by **patterns** that increase the probability to identify a data race

’11 Relaxer

- detects potential **data races** in sequentially consistent execution trace
- computes the set of potential **happens-before cycles** == possible violations of sequential consistency
- uses detected races to predict **alternative interleavings on a relaxed memory model**
- exploits **biased-random scheduler** to force the occurrence of such interleavings

---

Complementing with TC Generation

data races

[Narada OOPSLA’14]

atomicity violations

[Intruder PLDI’15]

deadlocks

[Omen FSE’15]

Complementing analysis with test case generation

’14 Narada

- monitors execution of sequential test suite with **lockset** analysis
- identifies **unprotected accesses** to shared elements, and infers state and invocation sequences that trigger data races
- **synthesises concurrent test cases** to expose the data race

’15 Intruder

- executes sequential test suite to **profile** the lock acquisitions, lock releases, field accesses
- infer possible atomicity violations with **lock-based analysis**
  based on four **memory access patterns** known to be **non-serializable**.
- combines sequential test cases to **generate concurrent test cases** that expose atomicity violations

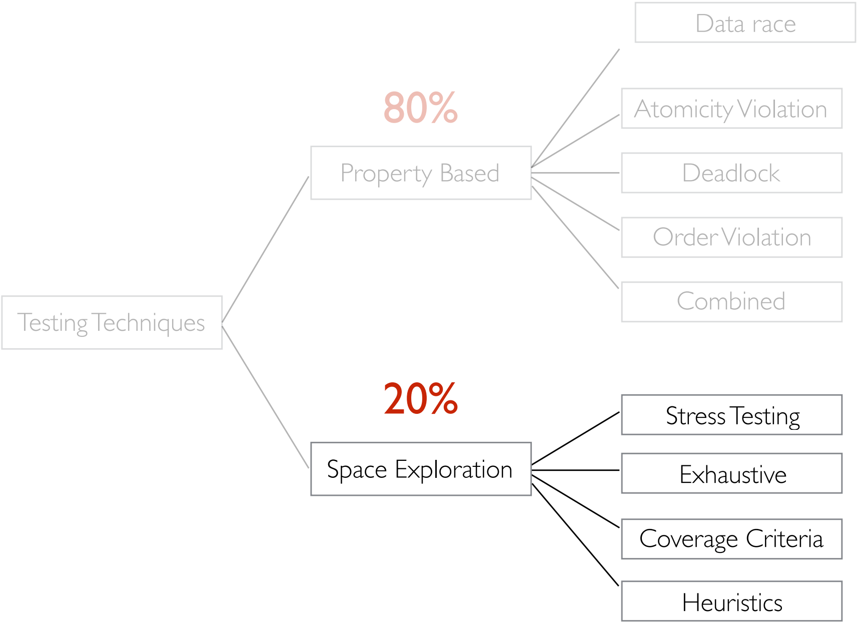
’15 OMEN

- reveals **deadlocks** by exploiting properties of sequential executions
- executes a **sequential test suite**
  builds a lock **dependency relation** that captures the lock acquisitions of the executed methods
- **generates concurrent test cases** from sequential ones

Correctness violations

concurrent behaviors that violate program specifications	typestate faults	order violations constraint solver
[JPredictor ICSE’08]	[Pretex ASE’08]	[ExceptioNull FSE’12]
[GPredict ICSE’15]	[2ndStrike ASPLOS’11]	

State exploration



---

violations of program specifications

jPredictor

- shrinks an execution trace to only events relevant for the property to be checked with **static analysis**
- builds a **causality graph** involving the selected events based on the notion of sliced causality (happens-before relation)
- predicts and executes **alternative interleavings** that might lead to property violations

GPredict

- verifies high level properties expressed as **regular expressions on the order of statements**
- infers the **order relations between events** dynamically identified on execution traces relying on thread-local traces, and ignoring global synchronisations,
- checks for the feasibility of interleavings that violate the concurrency properties by means of a **constraint solver** to predict possible concurrency faults

typestate faults

Pretex

- **typestate** == state associated with an object — set of operations that can be applied to the object in that state
- **typestate fault** == invoking an operation on an object obj in a typestate that does not support that operation

  (related to **high level semantics** of the target system)
- computes the **happens-before** relation among events
- determines which objects are shared
- infers typestate properties of each shared object relying on mining techniques
- generates a finite state machine model of the concurrent execution
- checks the generated model for typestate property violations

2nd-Strike

- detects **concurrency typestate** *faults* that involve **files**, **pointers locks**
- **dynamically analyzes** a test case execution to generate a set of candidate faults
- identify operations that cannot be reordered with **happens-before** relation
- uses a **deterministic scheduler** to force the execution of the candidate faults computed during the analysis

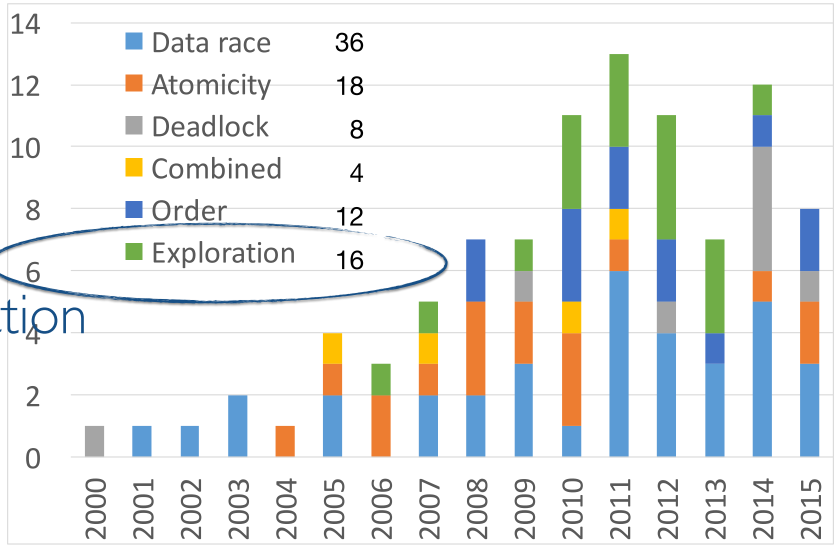
---

order violations with constraint solver

ExceptioNull

- detects interleavings that can lead to **null pointer dereferences of shared data items** with hybrid **lockset** and **happens-before** analysis

Research focus



space exploration

bounded state space exploration

stress testing	limit the amount of interleavings randomly
exhaustive (bounded) exploration	limit the depth of the interleavings
coverage of (property-relevant) interleavings	limit accord to the structure
heuristic-driven exploration	heuristic priority

Pruning the Interleaving Space

Model checking	systematic exploration of interleavings	exploration of data flow relations
[Joshi FSE'10]	[Wang ICSE'11]	[Tasharofi ASE'13]

SO FAR	OPEN ISSUES
selecting interleavings	generating test cases
property based approaches	
low level order violations	high level order violations
shared memory systems	message passing systems

reference

Francesco A. Bianchi, Alessandro Margara, Mauro Pezzè
A Survey of Recent Trends in Testing Concurrent Software Systems
IEEE Transactions on Software Engineering, May 2017

**Software
Testing and
Analysis
Research group**

L
T
A

Testing

GUI testing

Concurrent testing

Test oracles

Symbolic execution

field testing

cloud testing

ULS testing

Self healing

failure prediction

fault localisation

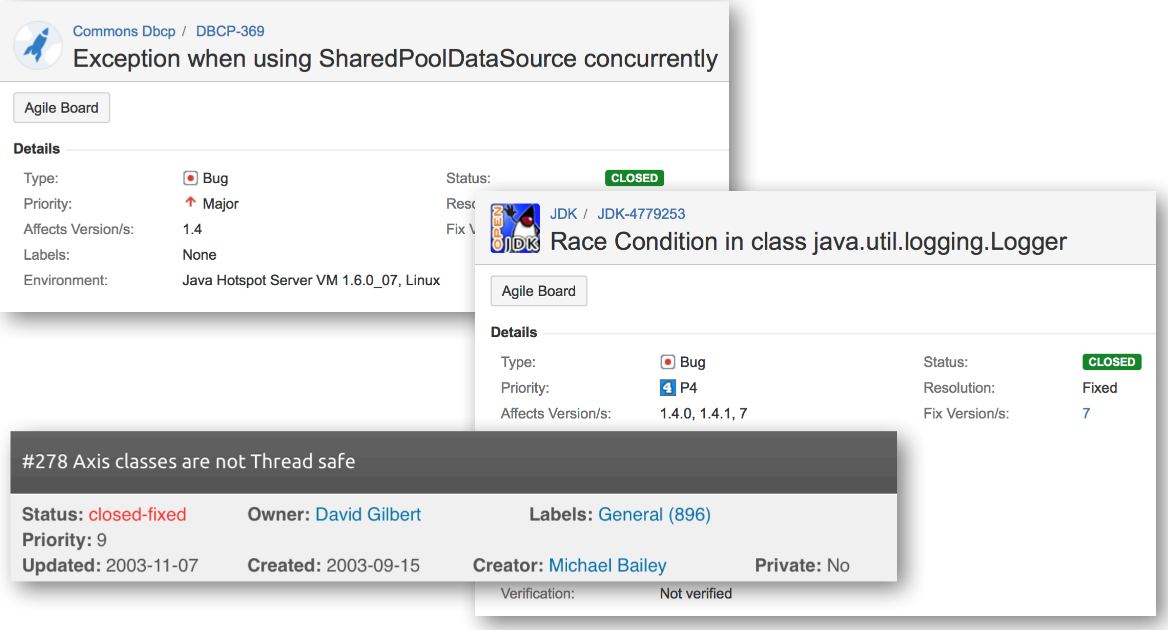
healing alerts

dynamic analysis

automated healing

Reproducing Concurrency Failures from Crash Stacks

Concurrency field failures



Commons Dbcp / DBCP-369

Exception when using SharedPoolDataSource concurrently

Agile Board

Details

Type: Bug

Priority: ↑ Major

Affects Version/s: 1.4

Labels: None

Environment: Java Hotspot Server VM 1.6.0_07, Linux

Status: CLOSED

Reso

Fix V

JDK / JDK-4779253

Race Condition in class java.util.logging.Logger

Agile Board

Details

Type: Bug

Priority: 4 P4

Affects Version/s: 1.4.0, 1.4.1, 7

Status: CLOSED

Resolution: Fixed

Fix Version/s: 7

Verification: Not verified

#278 Axis classes are not Thread safe

Status: closed-fixed **Owner:** David Gilbert **Labels:** General (896)

Priority: 9

Updated: 2003-11-07 **Created:** 2003-09-15 **Creator:** Michael Bailey **Private:** No

Failure inducing test code

test code **+** ***thread interleaving***

Executable code fragment
that exercises the program under test

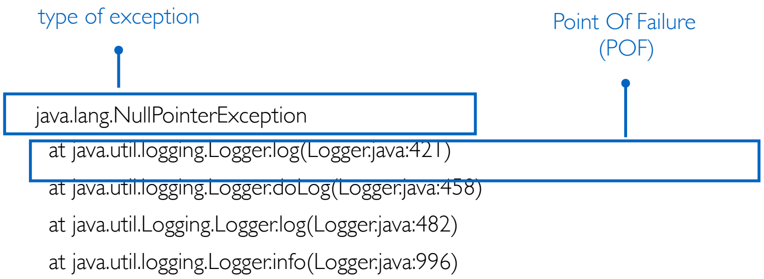
order of shared memory
accesses

Thread-safe Class

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

Crash Stack

JDK-4779253 : Race Condition in class java.util.logging.Logger

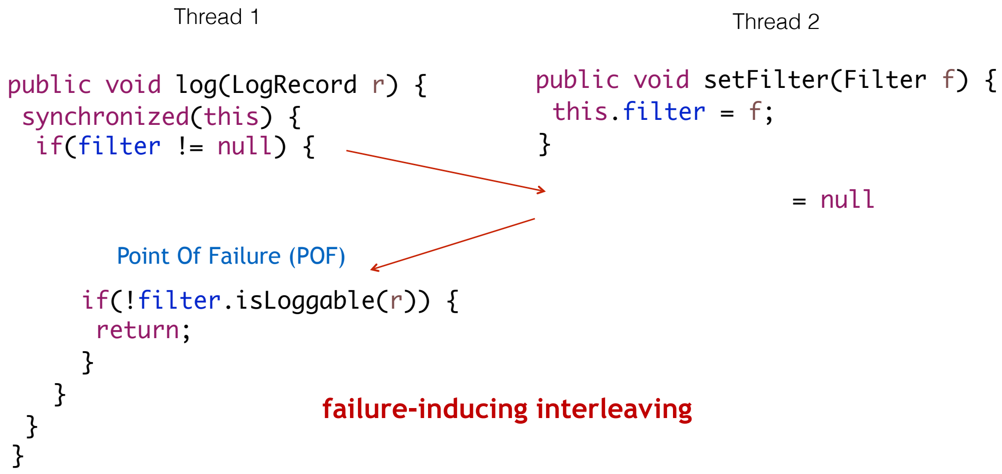


```
java.lang.NullPointerException
   at java.util.logging.Logger.log(Logger.java:421)
   at java.util.logging.Logger.doLog(Logger.java:458)
   at java.util.Logging.Logger.log(Logger.java:482)
   at java.util.logging.Logger.info(Logger.java:996)
```

"70% of concurrency failures lead to crashes or hangs"
Lu et al. ASPLOS '08

Threat-safety Violation

JDK-4779253 : Race Condition in class java.util.logging.Logger



Thread 1

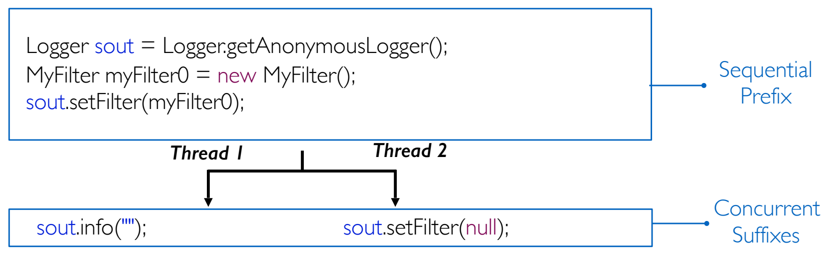
```
public void log(LogRecord r) {
 synchronized(this) {
  if(filter != null) {


     if(!filter.isLoggable(r)) {
      return;
     }
    }
   }
  }
```

Thread 2

```
public void setFilter(Filter f) {
 this.filter = f;
}

                  = null
```

Concurrent Test Code

JDK-4779253 : Race Condition in class java.util.logging.Logger



```
Logger sout = Logger.getAnonymousLogger();
MyFilter myFilter0 = new MyFilter();
sout.setFilter(myFilter0);
```

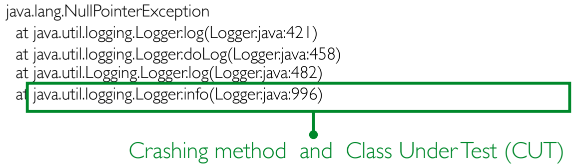
Thread 1	*Thread 2*

```
sout.info("");                              sout.setFilter(null);
```

Failure-inducing test code

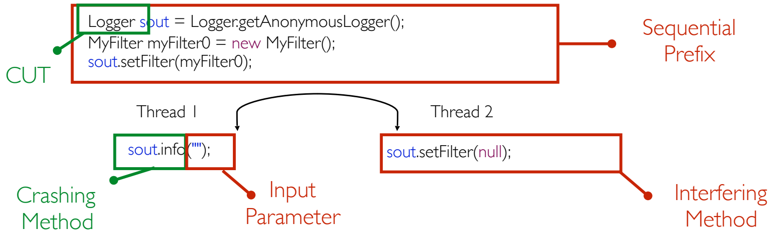
Limited information from Crash Stacks

Crash Stack



```
java.lang.NullPointerException
  at java.util.logging.Logger.log(Logger.java:421)
  at java.util.logging.Logger.doLog(Logger.java:458)
  at java.util.Logging.Logger.log(Logger.java:482)
  at java.util.logging.Logger.info(Logger.java:996)
```

Failure-inducing Test Code

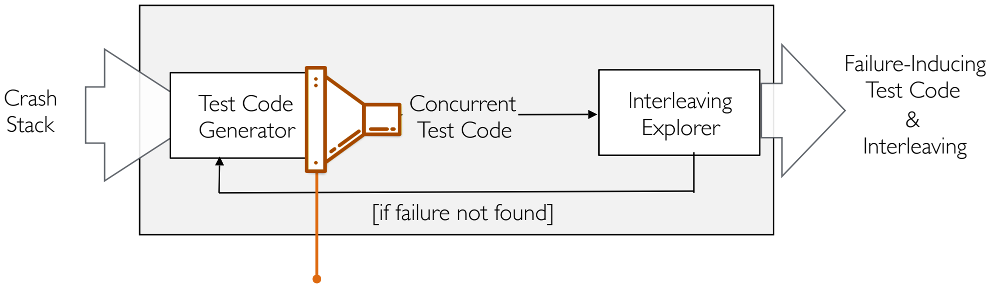


```
Logger sout = Logger.getAnonymousLogger();
MyFilter myFilter0 = new MyFilter();
sout.setFilter(myFilter0);
```

Thread 1	Thread 2

```
sout.info("");                  sout.setFilter(null);
```

ConCrash



Crash Stack

Test Code Generator

Concurrent Test Code

Interleaving Explorer

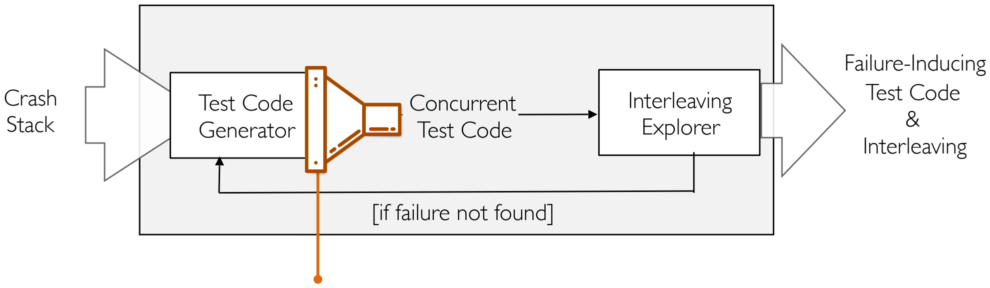
[if failure not found]

Failure-Inducing Test Code & Interleaving

Pruning Strategies

Avoid exploring the interleaving space of
redundant and **irrelevant** test codes

Pruning Strategies



Crash Stack

Test Code Generator

Concurrent Test Code

Interleaving Explorer

[if failure not found]

Failure-Inducing Test Code & Interleaving

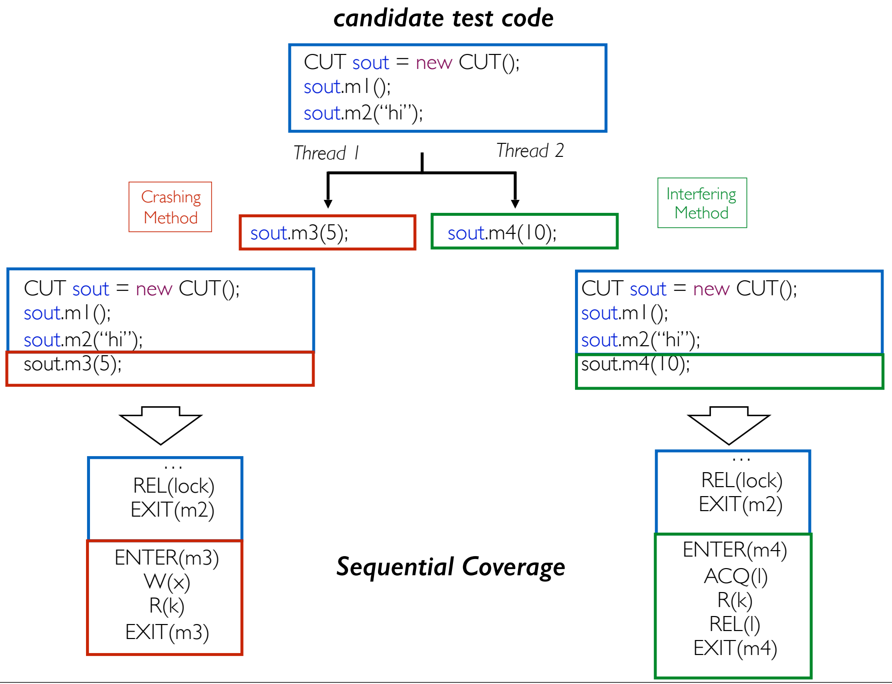
Pruning Strategies

Pruning Strategies

Uses information from executing call of a test code sequentially

Low computational cost

Pruning Strategies



candidate test code

```
CUT sout = new CUT();
sout.m1();
sout.m2("hi");
```

Thread 1

Thread 2

Crashing Method

Interfering Method

```
sout.m3(5);
```

```
sout.m4(10);
```

```
CUT sout = new CUT();
sout.m1();
sout.m2("hi");
sout.m3(5);
```

```
CUT sout = new CUT();
sout.m1();
sout.m2("hi");
sout.m4(10);
```

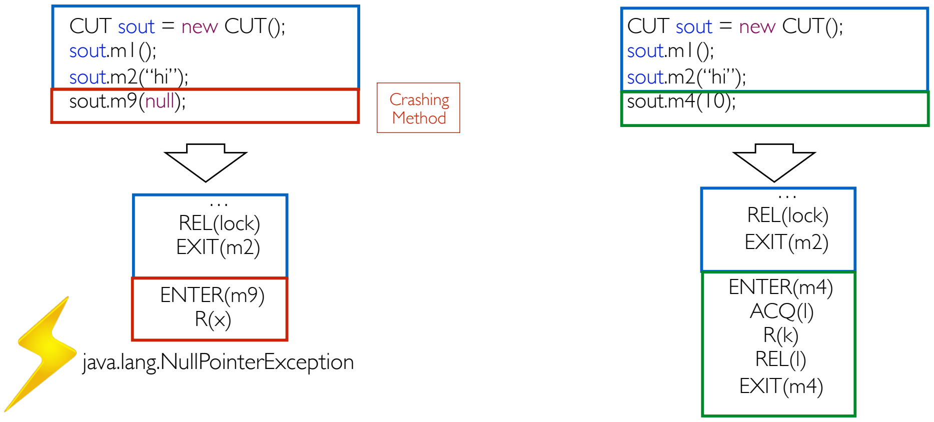
...
REL(lock)
EXIT(m2)
ENTER(m3)
W(x)
R(k)
EXIT(m3)

Sequential Coverage

...
REL(lock)
EXIT(m2)
ENTER(m4)
ACQ(l)
R(k)
REL(l)
EXIT(m4)

PS-Exception

one of its method call sequences throws an exception sequentially



```
CUT sout = new CUT();
sout.m1();
sout.m2("hi");
sout.m9(null);
```

Crashing Method

```
...
REL(lock)
EXIT(m2)
ENTER(m9)
R(x)
```

java.lang.NullPointerException

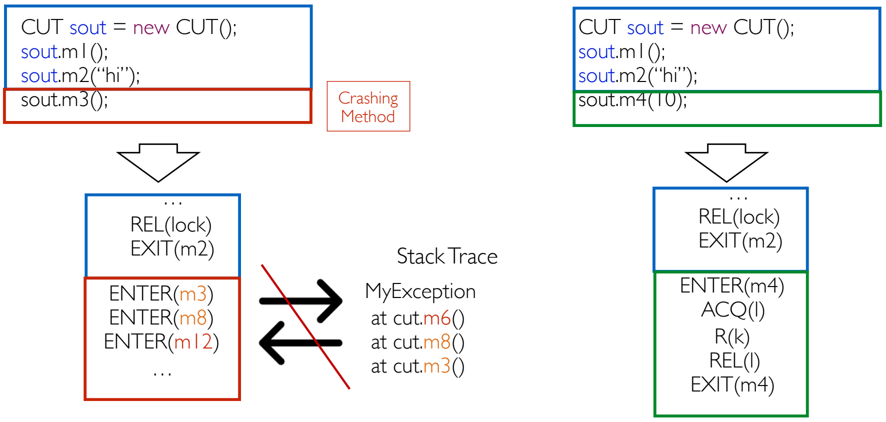
```
CUT sout = new CUT();
sout.m1();
sout.m2("hi");
sout.m4(10);
```

```
...
REL(lock)
EXIT(m2)
ENTER(m4)
ACQ(l)
R(k)
REL(l)
EXIT(m4)
```

Our focus are concurrent (not sequential) failures!

PS-Stack

Prunes a candidate test code if
the sequential coverage of the crashing method does not match the crash stack



```
CUT sout = new CUT();
sout.m1();
sout.m2("hi");
sout.m3();
```

Crashing Method

```
...
REL(lock)
EXIT(m2)
ENTER(m3)
ENTER(m8)
ENTER(m12)
...
```

Stack Trace

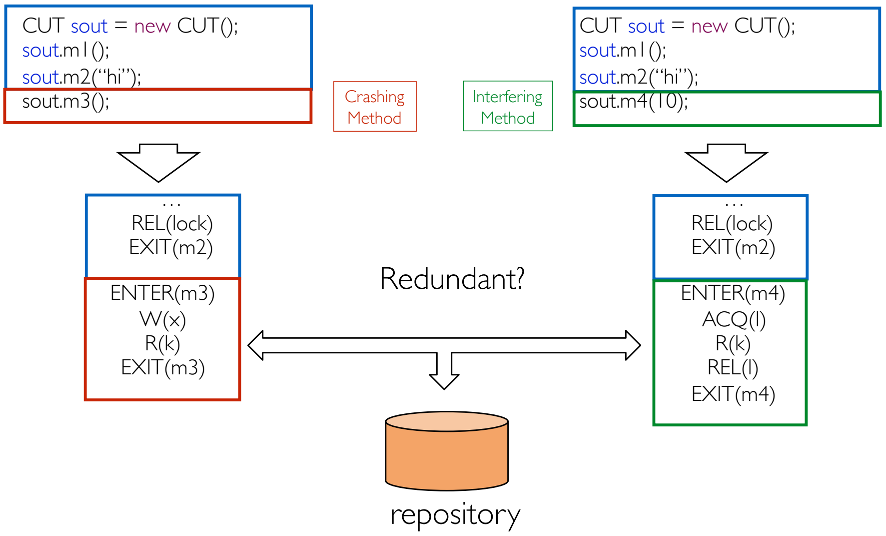
```
MyException
 at cut.m6()
 at cut.m8()
 at cut.m3()
```

```
CUT sout = new CUT();
sout.m1();
sout.m2("hi");
sout.m4(10);
```

```
...
REL(lock)
EXIT(m2)
ENTER(m4)
ACQ(l)
R(k)
REL(l)
EXIT(m4)
```

PS-Redundant

Prunes a candidate test code if
the sequential coverages of the concurrent suffixes are redundant



```
CUT sout = new CUT();
sout.m1();
sout.m2("hi");
sout.m3();
```

Crashing Method

Interfering Method

```
CUT sout = new CUT();
sout.m1();
sout.m2("hi");
sout.m4(10);
```

…
REL(lock)
EXIT(m2)
ENTER(m3)
W(x)
R(k)
EXIT(m3)

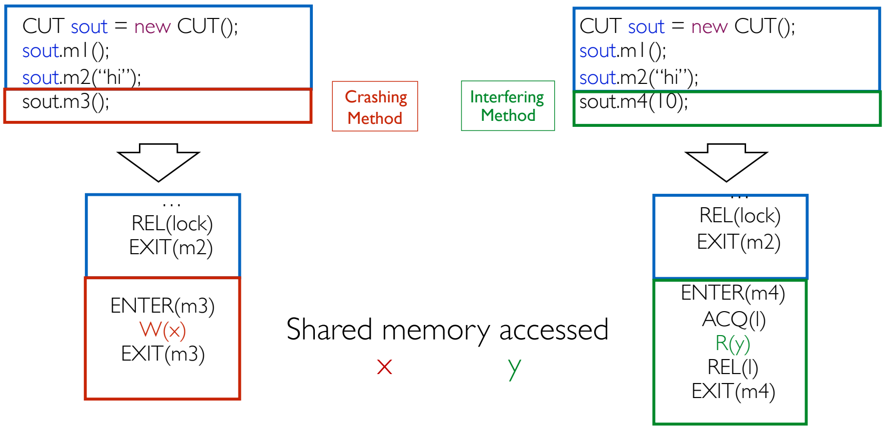
Redundant?

…
REL(lock)
EXIT(m2)
ENTER(m4)
ACQ(l)
R(k)
REL(l)
EXIT(m4)

repository

PS-Infere

Prunes a candidate test code if
the concurrent suffixes do not write- access the same shared memory location



```
CUT sout = new CUT();
sout.m1();
sout.m2("hi");
sout.m3();
```

Crashing Method

Interfering Method

```
CUT sout = new CUT();
sout.m1();
sout.m2("hi");
sout.m4(10);
```

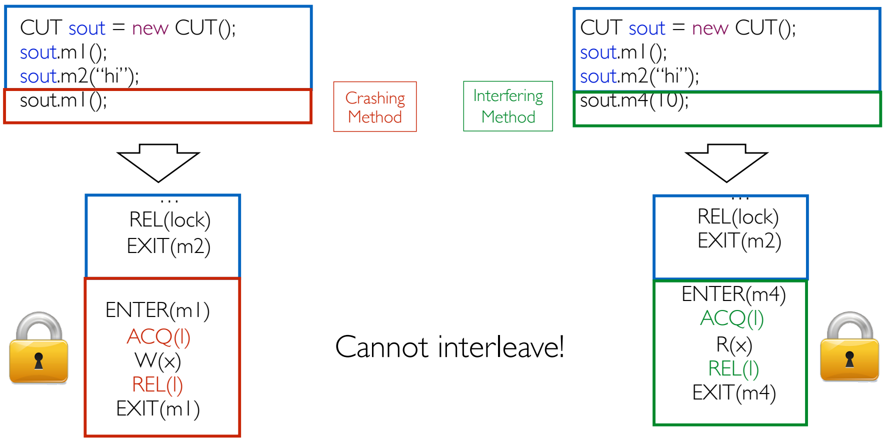
…
REL(lock)
EXIT(m2)
ENTER(m3)
W(x)
EXIT(m3)

Shared memory accessed
x y

…
REL(lock)
EXIT(m2)
ENTER(m4)
ACQ(l)
R(y)
REL(l)
EXIT(m4)

PS-Interleave

Prunes a candidate test code if
the concurrent suffixes are mutually exclusive

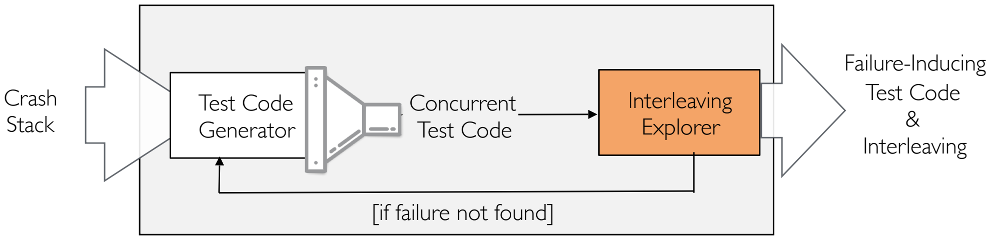


```
CUT sout = new CUT();
sout.m1();
sout.m2("hi");
sout.m1();
```

```
CUT sout = new CUT();
sout.m1();
sout.m2("hi");
sout.m4(10);
```

Cannot interleave!

Interleaving Explorer



Uses symbolic execution and constraint solving to identify failure inducing interleavings

Failure Reproduction

Failures reproduced in all runs

Class Under Test	Success Rate
PerUserPoolDataSource	100%
SharedPoolDataSource	100%
IntRange	100%
BufferedInputStream	100%
Logger	100%
PushbackReader	100%
NumberAxis	100%
XYSeries	100%
Category	100%
FileAppender	100%
AVG	100%

* Average results of 5 runs with a time budget of 5 hours

Reproduction Costs

Average failure reproduction time is less than 1 minute

Class Under Test	Success Rate	Failure Reprod.Time (sec)
PerUserPoolDataSource	100%	63
SharedPoolDataSource	100%	42
IntRange	100%	13
BufferedInputStream	100%	15
Logger	100%	70
PushbackReader	100%	7
NumberAxis	100%	30
XYSeries	100%	107
Category	100%	25
FileAppender	100%	92
AVG	100%	46

* Average results of 5 runs with a time budget of 5 hours

Generated test suite size

Effective test code generation

Class Under Test	Success Rate	Failure Reprod. Time (sec)	# Tests Retained after Pruning
PerUserPoolDataSource	100%	63	2
SharedPoolDataSource	100%	42	2
IntRange	100%	13	1
BufferedInputStream	100%	15	2
Logger	100%	70	3
PushbackReader	100%	7	1
NumberAxis	100%	30	1
XYSeries	100%	107	8
Category	100%	25	1
FileAppender	100%	92	5
AVG	100%	46	3

* Average results of 5 runs with a time budget of 5 hours

Generated test suite size

Small test codes

Class Under Test	**Success Rate**	**Failure Reprod. Time (sec)**	**# Tests Retained after Pruning**	**Test Size (# method calls)**
PerUserPoolDataSource	100%	63	2	4
SharedPoolDataSource	100%	42	2	4
IntRange	100%	13	1	4
BufferedInputStream	100%	15	2	5
Logger	100%	70	3	5
PushbackReader	100%	7	1	4
NumberAxis	100%	30	1	3
XYSeries	100%	107	8	6
Category	100%	25	1	5
FileAppender	100%	92	5	10
AVG	**100%**	**46**	**3**	**5**

* Average results of 5 runs with a time budget of 5 hours

Alternative approaches

ConTeGe	[Pradel and Gross PLDI '12] (random-based)
AutoConTest	[Terragni and Cheung ICSE '16] (coverage-based)

	ConTeGe		AutoConTest	
Class Under Test	Success Rate	Failure Reprod. Time (sec)	Success Rate	Failure Reprod. Time (sec)
PerUserPoolDataSource	0%	>18,000	0%	>18,000
SharedPoolDataSource	0%	>18,000	0%	>18,000
IntRange	0%	>18,000	100%	23
BufferedInputStream	80%	4,487	0%	>18,000
Logger	0%	>18,000	0%	>18,000
PushbackReader	20%	5,796	-	-
NumberAxis	0%	>18,000	100%	93
XYSeries	40%	12,387	0%	>18,000
Category	100%	14,410	-	-
FileAppender	0%	>18,000	-	-

---

reference

Francesco A. Bianchi, Mauro Pezzè, Valerio Terragni
Reproducing concurrency failures from crash stacks.
ESEC/SIGSOFT FSE 2017: 705-716