Testing Concurrent and Distributed Systems

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

• 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

Initially balance ≥ 0

Preserves balance ≥ 0

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

serial execution preserves invariants

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

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

concurrent execution and race conditions

race condition on balance

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

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
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

serializability

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
def withdraw(amt):
    balance = balance - amt
    return amt

def withdraw(amt):
    balance = balance - amt
    return amt

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)

1. class Value {
2.     private int x = 1;
3.     public synchronized void add(Value v) { x = x + v.get(); }
4.     public int get() { return x; }
5. }

6. class Task extends Thread {
7.     Value v1; Value v2;
8.     public Task(Value v1, Value v2) {
9.         this.v1 = v1; this.v2 = v2;
10.        this.start();
11.    }
12.    public void run() { v1.add(v2); }
13. }

14. class Main {
15.     public static void main(String[] args) {
16.         Value v1 = new Value(); Value v2 = new Value();
17.         new Task(v1, v2); new Task(v2, v1);
18.     }
19. }

20. a data race ...

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
1. **class** Value {
2.    **private** int x = 1;
3. 4.    **public synchronized** void add(Value v){x = x + v.get();}
5. 6.    **public synchronized** 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. }

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

---

1. **class** Value {
2.    **private** int x = 1;
3. 4.    **public synchronized** void add(Value v){x = x + v.get();}
5. 6.    **public synchronized** 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. }
public class Logger{
    ...
    private Filter filter;
    //Thread 1
    public void log(LogRecord record){
        if (record.getLevel().intValue() < levelValue
            || levelValue == offValue) {
                return;
        }
        synchronized (this) {
            if (filter != null){
                if (!filter.isLoggable(record)) {
                    return;
                }
            }
            ...
        }
        // Thread 2
        public void setFilter(Filter f) {
            this.filter = f;
        }
    }
}

but may fail

limit concurrency to prevent data races

locks (Java Synchronized)
public class Logger{

...  
private Filter filter;
//Thread 1
public void log(LogRecord record){
   if (record.getLevel().intValue() < levelValue || levelValue == offValue) {
      return;
   }
   synchronized (this) {  
      if (filter != null){
         if (!filter.isLoggable(record)) {
            return;
         }
      }
   }   // Thread 2
public void setFilter(Filter f) {
   this.filter = f;
}
"
"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.

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]
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

\[
\begin{align*}
W_1(y) & \quad W_2(t) & \quad R_1(x) & \quad R_1(x) & \quad W_2(x) & \quad R_1(y) & \quad R_2(t) \\
W_2(t) & \quad W_1(y) & \quad R_1(x) & \quad R_1(x) & \quad W_2(x) & \quad R_1(y) & \quad R_2(t) \\
W_1(y) & \quad W_2(t) & \quad R_1(x) & \quad W_2(x) & \quad R_1(x) & \quad R_1(y) & \quad R_2(t) \\
\vdots & & & & & & \\
W_1(y) & \quad W_2(t) & \quad R_1(x) & \quad R_1(x) & \quad W_2(x) & \quad R_2(t) & \quad R_1(y) \\
\end{align*}
\]

interleavings

Oracle

\[
<\text{input, interleaving, oracle}>
\]
testing concurrent systems

- System Model
- Generating Test Cases
- Selecting Interleavings
- Execute
- Output

Property of interleavings (data race, deadlock, …)

main focus of research

- System Model
- Generating Test Cases
- Test Cases
- Selecting Interleavings
- Execute
- Output

Property of interleavings (data race, deadlock, …)
Property based

INPUT

Test Case

T1
R(x) → W(x)

T2
R(x) → W(x)

W1(y) W2(t) R1(x) R1(x) W2(x) R1(y) R2(t)

W2(t) W1(y) R1(x) R1(x) W2(x) R1(y) R2(t)

W1(y) W2(t) R1(x) W2(x) R1(x) R1(y) R2(t)

... ...

W1(y) W2(t) R1(x) R1(x) W2(x) R2(t) R1(y)

taxonomy

Techniques

Property Based

Data race
Atomicity Violation
Deadlock
Order Violation
Combined

Space Exploration

Stress Testing
Exhaustive
Coverage Criteria
Heuristics
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

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) = \emptyset

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

**Example**: LockSet(A) \cap LockSet(B) \cap LockSet(C) = \emptyset

The public class `Logger`:

```java
public class Logger{
...
private Filter filter;
//Thread 1
public void log(LogRecord record){
    if (record.getLevel().intValue() < levelValue
        || levelValue == offValue) {
        return;
    }
    synchronized (this) {
        if (filter != null){
            if (!filter.isLoggable(record)) {
                return;
            }
        }.
    }
    ...
// Thread 2
public void setFilter(Filter f) {
    this.filter = f;
}
```

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) = \emptyset

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

**Example**: LockSet(A) \cap LockSet(B) \cap LockSet(C) = \emptyset

- Thread 1
  - A:Read(filter)
  - B:Read(filter)
  - Unlock(this)

- Thread 2
  - C:Write(filter)
  - A:Read(filter)
  - C:Write(filter)

Race Condition
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

<table>
<thead>
<tr>
<th>Thread</th>
<th>Program trace</th>
<th>Locks held</th>
<th>Lockset(x)</th>
</tr>
</thead>
<tbody>
<tr>
<td>thread A</td>
<td>lock(lck1)</td>
<td>{}</td>
<td>{lck1, lck2}</td>
</tr>
<tr>
<td></td>
<td>x=x+1</td>
<td>{……………..}</td>
<td></td>
</tr>
<tr>
<td></td>
<td>unlock(lck1)</td>
<td>{……………..}</td>
<td>{……………..}</td>
</tr>
<tr>
<td>thread B</td>
<td>lock{lck2}</td>
<td>{……………..}</td>
<td></td>
</tr>
<tr>
<td></td>
<td>x=x+1</td>
<td>{……………..}</td>
<td></td>
</tr>
<tr>
<td></td>
<td>unlock(lck2)</td>
<td>{……………..}</td>
<td>{……………..}</td>
</tr>
</tbody>
</table>

**INIT:** all locks for x

```
\{lck1\}  \{lck1\}
\{lck2\}  \{lck2\}
```

Intersect with locks held

lck1 held

lck2 held

Empty intersection

potential race
public class Logger{
    ...
    private Filter filter;
    //Thread 1
    public void log(LogRecord record){
        if (record.getLevel().intValue() < levelValue || levelValue == offValue) {
            return;
        }
        synchronized (this) {
            if (filter != null) {
                if (!filter.isLoggable(record)) {
                    return;
                }
            }
        }
    }
    // Thread 2
    public void setFilter(Filter f) {
        this.filter = f;
    }
}
Java.util.loggin.Logger

Lock(this)
LockSet(A) = {this}
LockSet(B) = {this}
LockSet(C) = Ø
LockSet(A) ∩ LockSet(B) ∩ LockSet(C) = Ø
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

![Diagram showing state transitions and lock acquisition](image)

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) =>
            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 =>
            send(Write)
            writer ! Write(name)
            terminator ! ActionDone
    }
}

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

Msg Write is received by Writer and the append method is fine
Msg Flush is received by Writer results is set to null

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 $i < j$
- $e_i = \text{send}(msg_k)$ and $e_j = \text{receive}(msg_k)$
  (a message is always sent before being received)
Happens-before relations:

- \( \text{send}(\text{Write}) < \text{send}(\text{ActionDone}) \) (intra thread)
- \( \text{send}(\text{Write}) < \text{receive}(\text{Write}) \) (inter thread)
- \( \text{send}(\text{ActionDone}) < \text{receive}(\text{ActionDone}) \) (intra thread)
- \( \text{receive}(\text{ActionDone}) < \text{send}(\text{Flush}) \) (intra thread)
- \( \text{send}(\text{Flush}) < \text{receive}(\text{Flush}) \) (intra thread)

Concurrent events:

- \( \text{receive}(\text{ActionDone}) \) and \( \text{receive}(\text{Write}) \)
- \( \text{send}(\text{Flush}) \) and \( \text{receive}(\text{Write}) \)
- \( \text{receive}(\text{Write}) \) and \( \text{receive}(\text{Flush}) \)

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 \( i < j \)
  - \( e_i = \text{send}(msg_k) \) and \( e_j = \text{receive}(msg_k) \)
  - (a message is always sent before being received)

Happens-before relations:

- \( \text{send}(\text{Write}) < \text{send}(\text{ActionDone}) \) (intra thread)
- \( \text{send}(\text{Write}) < \text{receive}(\text{Write}) \) (inter thread)
- \( \text{send}(\text{ActionDone}) < \text{receive}(\text{ActionDone}) \) (inter thread)
- \( \text{receive}(\text{ActionDone}) < \text{send}(\text{Flush}) \) (intra thread)
- \( \text{send}(\text{Flush}) < \text{receive}(\text{Flush}) \) (inter thread)

Concurrent events:

- \( \text{receive}(\text{ActionDone}) \) and \( \text{receive}(\text{Write}) \)
- \( \text{send}(\text{Flush}) \) and \( \text{receive}(\text{Write}) \)
- \( \text{receive}(\text{Write}) \) and \( \text{receive}(\text{Flush}) \)
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)
      terminaor ! ActionDone
    } send(Write) send(ActionDone)
  }
}

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

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 $i < j$
- $e_i = send(msg_i)$ and $e_j = receive(msg_j)$ (a message is always sent before being received)

**class** Writer extends Actor {
  var results = ArrayBuffer[String]()
  def receive() = {
    case Write(result:String) =>
      results.append(result)
      receive(Write)
    case Flush => {
      writeToExternal(results)
      results = null
      receive(Flush)
    }
  }
}
**class** Action(name:String, terminator:Terminator, writer:Writer) extends Actor {
  def receive() = {
    case Execute => {
      writer ! Write(name)
      terminator ! ActionDone
      send(Write)
      send(ActionDone)
    }
  }
}
**class** Terminator(actionNum:Int, writer:Writer) extends Actor {
  var curActions = actionNum
  def receive() = {
    case ActionDone => {
      curActions -= 1
      if (curActions == 0) writer ! Flush
      receive(ActionDone)
      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)
      results = null
    writeToExternal(results)
  }
}

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

Msg Flush is received by Writer and results is set to null
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
1. class Value {
2.     private int x = 1;
3.     public synchronized void add(Value v){x = x + v.get();}
4.     public synchronized int get(){return x;}
5. }
6. class Task extends Thread{
7.     Value v1; Value v2;
8.     public Task(Value v1,Value v2){
9.         this.v1 = v1; this.v2 = v2;
10.        this.start();
11.     }
12.     public void run(){v1.add(v2);}
13. }
14. class Main{
15.     public static void main(String[] args){
16.         Value v1 = new Value(); Value v2 = new Value();
17.         new Task(v1,v2); new Task(v2,v1);
18.     }
19. }

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

potential deadlock:
• Task T1 locks V1
• Task T2 locks V2
• Task T1 waits for V2
• Task T2 waits for V1

Goodlock algorithm

AT RUNTIME:
record the locking pattern for each thread during runtime as a lock tree
one lock tree per thread == 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 <t₁, t₂> and each operation on a shared memory location n₁ of t₁
check that no lock below n₁ in t₁ is above a node n₂ in a thread t₂
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

Systems
SOSP, ASPLOS, ISCA
15/94

Formal Methods
CAV, SPIN
2/94

Software Engineering
FSE, ICSE, ISSTA, ASE
42/94

Programming Languages
OOPSLA, PPOP, PLDI
42/94
Research focus

The diagram shows the research focus from 2000 to 2015. The categories include Data race, Atomicity, Deadlock, Combined, Order, and Exploration. The number of publications for each category over the years is indicated.

- **Data race**: 36 publications
- **Atomicity**: 18 publications
- **Deadlock**: 8 publications
- **Combined**: 4 publications
- **Order**: 12 publications
- **Exploration**: 16 publications

The image highlights the trend and focus areas in research over the last decade.
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**

[Velodrome PLDI'08]

[AtomFuzzer FSE'08]

[Penelope FSE'10]

**data races**

[RaceFuzzer PLDI'08]

[Frost SOSP'11]

[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

<table>
<thead>
<tr>
<th>data races</th>
<th>atomicity violations</th>
<th>deadlocks</th>
</tr>
</thead>
<tbody>
<tr>
<td>[FastTrack PLDI’09]</td>
<td>[Falcon ICSE’10]</td>
<td>[MagicFuzzer ICSE’12]</td>
</tr>
<tr>
<td>[LiteRace PLDI’09]</td>
<td></td>
<td>[ConLock ICSE’14]</td>
</tr>
<tr>
<td>[Carisma ISSTA’12]</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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]

Problems:

- 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

<table>
<thead>
<tr>
<th></th>
<th>Web</th>
<th>Event-based</th>
<th>Android</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>[WebRacer PLDI’12]</td>
<td>[EventRacer OOPSLA’13]</td>
<td>[DroidRacer PLDI’14]</td>
</tr>
</tbody>
</table>
new paradigms

\textbf{'12 WebRacer}
- \textbf{happens-before} analysis enhanced with the semantics of \textit{Web} platforms — focus on
  - variable races == \textbf{data races} caused by \textit{concurrent accesses to shared memory locations}
  - HTML races == \textbf{accesses of DOM nodes} may occur both \textbf{before} and \textbf{after creations}
  - function races == \textbf{function invocations} occur both \textbf{before} and \textbf{after parsing} the functions
  - event dispatch races == \textbf{events} fire both \textbf{before} and \textbf{after adding the event handlers}

\textbf{'13 EventRacer}
- \textbf{happens-before analysis for event-based programs}

\textbf{'14 DroidRacer}
- exploits concurrency semantics of \textit{Android} programming model to derive precise \textbf{happens-before} relation
to reduces of false positives

happens-before for relaxed memory models

\begin{center}
\begin{tabular}{ccc}
\textbf{C++} & \textbf{Java} & \textbf{Android} \\
\end{tabular}
\end{center}
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

[GPredict ICSE'15] [Predictor ICSE'08] [ExceptioNull FSE'12]

typestate faults

[Pretex ASE'08] [2ndStrike ASPLOS'11]

order violations constraint solver
State exploration

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

80%

Property Based

20%

Stress Testing
- Exhaustive
- Coverage Criteria
- Heuristics

Testing Techniques

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

violations of program specifications
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

![Graph showing research focus from 2000 to 2015]

space exploration

bounded state space exploration

- stress testing
- exhaustive (bounded) exploration
- coverage of (property-relevant) interleavings
- heuristic-driven exploration

- limit the amount of interleavings randomly
- limit the depth of the interleavings
- limit accord to the structure
- heuristic priority
Pruning the Interleaving Space

**Model checking**  
selecting interleavings  
property based approaches  
low level order violations  
shared memory systems

**systematic exploration of interleavings**  
[Joshi FSE’10]

**exploration of data flow relations**  
[Wang ICSE’11]  
[Tasharofi ASE’13]

**SO FAR**

**OPEN ISSUES**

generating test cases  
high level order violations  
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

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
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**

```
public void log(LogRecord r) {
    synchronized(this) {
        if(filter != null) {
            if(!filter.isLoggable(r)) {
                return;
            }
        }
    }
}
```
Concurrent Test Code

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

Sequential Prefix

Concurrent Suffixes

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.LogRecordLog(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

Sequential Prefix

Interfering Method

Crashing Method

Input Parameter
ConCrash

Pruning Strategies
Avoid exploring the interleaving space of redundant and irrelevant test codes
Pruning Strategies

Uses information from executing call of a test code sequentially

Low computational cost
PS-Exception

one of its method call sequences throws an exception sequentially

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

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

Our focus are concurrent (not sequential) failures!

```
java.lang.NullPointerException
```

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

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();
```

```
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)

Stack Trace

REL(lock) EXIT(m2)
PS-Redundant

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

PS-Infere

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

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

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

Cannot interleave!

Interleaving Explorer

Uses symbolic execution and constraint solving to identify failure-inducing interleavings
## Failure Reproduction

*Failures reproduced in all runs*

<table>
<thead>
<tr>
<th>Class Under Test</th>
<th>Success Rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>PerUserPoolDataSource</td>
<td>100%</td>
</tr>
<tr>
<td>SharedPoolDataSource</td>
<td>100%</td>
</tr>
<tr>
<td>IntRange</td>
<td>100%</td>
</tr>
<tr>
<td>BufferedInputStream</td>
<td>100%</td>
</tr>
<tr>
<td>Logger</td>
<td>100%</td>
</tr>
<tr>
<td>PushbackReader</td>
<td>100%</td>
</tr>
<tr>
<td>NumberAxis</td>
<td>100%</td>
</tr>
<tr>
<td>XYSeries</td>
<td>100%</td>
</tr>
<tr>
<td>Category</td>
<td>100%</td>
</tr>
<tr>
<td>FileAppender</td>
<td>100%</td>
</tr>
<tr>
<td>AVG</td>
<td>100%</td>
</tr>
</tbody>
</table>

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

## Reproduction Costs

*Average failure reproduction time is less than 1 minute*

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

* Average results of 5 runs with a time budget of 5 hours
### Generated test suite size

**Effective test code generation**

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

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

### Generated test suite size

**Small test codes**

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

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

<table>
<thead>
<tr>
<th>Class Under Test</th>
<th>ConTeGe Success Rate</th>
<th>ConTeGe Reprod. Time (sec)</th>
<th>AutoConTest Success Rate</th>
<th>AutoConTest Reprod. Time (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>PerUserPoolDataSource</td>
<td>0%</td>
<td>&gt;18,000</td>
<td>0%</td>
<td>&gt;18,000</td>
</tr>
<tr>
<td>SharedPoolDataSource</td>
<td>0%</td>
<td>&gt;18,000</td>
<td>0%</td>
<td>&gt;18,000</td>
</tr>
<tr>
<td>IntRange</td>
<td>0%</td>
<td>&gt;18,000</td>
<td>100%</td>
<td>23</td>
</tr>
<tr>
<td>BufferedInputStream</td>
<td>80%</td>
<td>4,487</td>
<td>0%</td>
<td>&gt;18,000</td>
</tr>
<tr>
<td>Logger</td>
<td>0%</td>
<td>&gt;18,000</td>
<td>0%</td>
<td>&gt;18,000</td>
</tr>
<tr>
<td>PushbackReader</td>
<td>20%</td>
<td>5,796</td>
<td>100%</td>
<td>-</td>
</tr>
<tr>
<td>NumberAxis</td>
<td>0%</td>
<td>&gt;18,000</td>
<td>100%</td>
<td>93</td>
</tr>
<tr>
<td>XYSeries</td>
<td>40%</td>
<td>12,387</td>
<td>0%</td>
<td>&gt;18,000</td>
</tr>
<tr>
<td>Category</td>
<td>100%</td>
<td>14,410</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>FileAppender</td>
<td>0%</td>
<td>&gt;18,000</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

Alternative approaches

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

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