CAN WE MONITOR ALL MULTITHREADED PROGRAMS?
A tutorial Using RV tools for Java Programs

Antoine El-Hokayem and Yliès Falcone

Univ. Grenoble Alpes, Inria, LIG, CNRS

The 18th International Conference on Runtime Verification
Limassol, Cyprus

November 10, 2018
<table>
<thead>
<tr>
<th>Instructions</th>
<th>Introduction</th>
<th>Checking Sequences</th>
<th>Checking Concurrency</th>
<th>Multi-Trace</th>
<th>Conclusion</th>
</tr>
</thead>
</table>

**INSTRUCTIONS ➔ PREPARE THE TUTORIAL FILES**

1. The tutorial repository is hosted at [https://gitlab.inria.fr/monitoring/rv-multi](https://gitlab.inria.fr/monitoring/rv-multi)
2. Make sure to have docker installed (and running)
3. Set up the docker container (sudo is not needed if docker runs in userpace)

```
1. git clone https://gitlab.inria.fr/monitoring/rv-multi.git
2. cd rv-multi/docker
3. sudo make fetch
4. sudo make run
```

You should see:

```
— rv-multithreaded —
root's password is 'root'

Browse the README files:
http://localhost:8050/
```

```
[user@rv-multi rv-multi]$
```
RV & Multithreaded Programs
CONTEXT ➔ BRIEF OVERVIEW OF RV

- Lightweight verification technique
- Checks whether a run of a program conforms to a specification
  (As opposed to model checking which verifies all runs)
- The run is captured as a trace, typically seen as a sequence of events
CONTEXT ← BRIEF OVERVIEW OF RV

- **Lightweight** verification technique
- Checks whether a run of a program conforms to a specification (As opposed to model checking which verifies all runs)
- The run is captured as a trace, typically seen as a sequence of events

**Specification**

**Instrumentation**

**System**
CONTEXT → BRIEF OVERVIEW OF RV

- Lightweight verification technique
- Checks whether a run of a program conforms to a specification (As opposed to model checking which verifies all runs)
- The run is captured as a trace, typically seen as a sequence of events

### System

Instrumentation

Trace

Specification
• **Context → Brief Overview of RV**

  - **Lightweight** verification technique
  - Checks whether a **run** of a program conforms to a specification (As opposed to model checking which verifies all runs)
  - The run is **captured** as a trace, typically seen as a sequence of events

![Diagram](image-url)
**Context → Brief Overview of RV**

- **Lightweight** verification technique
- Checks whether a **run** of a program conforms to a specification (As opposed to model checking which verifies all runs)
- The run is **captured** as a trace, typically seen as a sequence of events
- **Monitors** are synthesized (and integrated) to observe the system
CONTEXT ← BRIEF OVERVIEW OF RV

- Lightweight verification technique
- Checks whether a run of a program conforms to a specification (As opposed to model checking which verifies all runs)
- The run is captured as a trace, typically seen as a sequence of events
- Monitors are synthesized (and integrated) to observe the system
- Monitors determine a verdict: $\mathbb{B}_3 = \{\top, \bot, ?\}$
  - $\top$ (true): run complies with specification
  - $\bot$ (false): run does not comply with specification
  - ?: verdict cannot be determined (yet)
Let us consider producer-consumer

- All threads access a shared queue
- Producers add items on the queue
- Consumers remove items from the queue
CONTEXT $\mapsto$ Example Multithreaded Program

- Let us consider producer-consumer
  - All threads access a shared queue
  - Producers add items on the queue
  - Consumers remove items from the queue

- A correct execution complies with the following properties:
  1. $(\varphi_1)$ Consumers must not remove an item unless the queue contains one
  2. $(\varphi_2)$ All items placed on the queue must be eventually consumed
**CONTEXT** ← **EXECUTIONS OF PRODUCER-CONSUMER**

1. `public class SynchQueue {`
2. `    private LinkedList<Integer> q = new LinkedList<Integer>();`
3. `    public void produce(Integer v) { q.add(v); }
4. `    public Integer consume() { return q.poll(); }
5. `}

A. El-Hokayem, Y. Falcone, Can We Monitor All Multithreaded Programs?
**CONTEXT ➔ EXECUTIONS OF PRODUCER-CONSUMER**

```java
public class SynchQueue {
    private LinkedList<Integer> q = new LinkedList<Integer>();
    public void produce(Integer v) { q.add(v); }
    public Integer consume() { return q.poll(); }
}
```

Thread 0 (Producer)

1. sq.produce(0);
2. sq.produce(1);

Thread 1 (Consumer)

1. sq.consume(); //0
2. sq.consume(); //1

A. El-Hokayem, Y. Falcone, Can We Monitor All Multithreaded Programs?
EXECUTIONS OF PRODUCER-CONSUMER

Thread 0 (Producer)

1 sq.produce(0);
2 sq.produce(1);

Thread 1 (Consumer)

2 sq.consume(); //0
4 sq.consume(); //1

Execution Verdict

1 2 3 4 ⊤
CONTEXT ← EXECUTIONS OF PRODUCER-CONSUMER

```java
public class SynchQueue {
    private LinkedList<Integer> q = new LinkedList<Integer>();
    public void produce(Integer v) { q.add(v); }
    public Integer consume() { return q.poll(); }
}
```

<table>
<thead>
<tr>
<th>Execution</th>
<th>Verdict</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 2 3 4</td>
<td>⊤</td>
</tr>
<tr>
<td>1 3 2 4</td>
<td>⊤</td>
</tr>
</tbody>
</table>

Thread 0 (Producer)

1. sq.produce(0);
2. sq.produce(1);

Thread 1 (Consumer)

1. sq.consume(); //0
2. sq.consume(); //1

Execution Verdict

- Consume on an empty queue ($\phi_1$)
- One element left in queue ($\phi_2$)
- Violates both ($\phi_1$) and ($\phi_2$)
### CONTEXT \(\rightarrow\) Executions of Producer-consumer

```java
public class SynchQueue {
    private LinkedList<Integer> q = new LinkedList<Integer>();
    public void produce(Integer v) { q.add(v); }
    public Integer consume() { return q.poll(); }
}
```

Thread 0 (Producer)

1. sq.produce(0);
2. sq.produce(1);

Thread 1 (Consumer)

1. sq.consume(); //0
2. sq.consume(); //1

<table>
<thead>
<tr>
<th>Execution</th>
<th>Verdict</th>
</tr>
</thead>
<tbody>
<tr>
<td>1234</td>
<td>⊤</td>
</tr>
<tr>
<td>1324</td>
<td>⊤</td>
</tr>
<tr>
<td>2134</td>
<td>⊥</td>
</tr>
</tbody>
</table>

Consume on an empty queue \((\varphi_1)\)
CONTEXT ↔ EXECUTIONS OF PRODUCER-CONSUMER

```java
public class SynchQueue {
    private LinkedList<Integer> q = new LinkedList<Integer>();
    public void produce(Integer v) { q.add(v); }
    public Integer consume() { return q.poll(); }
}
```

Thread 0 (Producer)
1. sq.produce(0);
2. sq.produce(1);

Thread 1 (Consumer)
1. sq.consume(); //0
2. sq.consume(); //1

Execution Verdict

- Consume on an empty queue ($\varphi_1$)
- One element left in queue ($\varphi_2$)
**CONTEXT** ← **EXECUTIONS OF PRODUCER-CONSUMER**

```java
public class SynchQueue {
    private LinkedList<Integer> q = new LinkedList<Integer>();
    public void produce(Integer v) { q.add(v); }
    public Integer consume() { return q.poll(); }
}
```

<table>
<thead>
<tr>
<th>Thread 0 (Producer)</th>
<th>Thread 1 (Consumer)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 sq.produce(0);</td>
<td>2 sq.consume(); //0</td>
</tr>
<tr>
<td>3 sq.produce(1);</td>
<td>4 sq.consume(); //1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Execution</th>
<th>Verdict</th>
</tr>
</thead>
<tbody>
<tr>
<td>1234</td>
<td>⊤</td>
</tr>
<tr>
<td>1324</td>
<td>⊤</td>
</tr>
<tr>
<td>2134</td>
<td>⊥, Consume on an empty queue ($\varphi_1$)</td>
</tr>
<tr>
<td>132</td>
<td>⊥, One element left in queue ($\varphi_2$)</td>
</tr>
<tr>
<td>2413</td>
<td>⊥, Violates both ($\varphi_1$) and ($\varphi_2$)</td>
</tr>
</tbody>
</table>

A. El-Hokayem, Y. Falcone, Can We Monitor All Multithreaded Programs?
An execution of a parallel program is best seen as a partial order (happens-before).

Typical RV formalisms operate on a total order (sequence) of events.

An instrumented program must capture the order of events as it happens during the execution to pass it to monitors.

Consistent with weak memory consistency models \([AG96, ANB + 95, MPA05]\), Mazurkiewicz traces \([Maz86, GK10]\), parallel series \([LW01]\), Message Sequence Charts graphs \([MR04]\), and Petri Nets \([NPW81]\).

LTL, MTL \([TR05]\), CFG, ERE, QEA \([RCR15]\), DATE \([CPS09]\), LTL monitors \([BLS11]\).
• An execution of a parallel program is best seen as a partial order (happens-before)\(^1\).

---

\(^1\) Consistent with weak memory consistency models [AG96, ANB\(^+\)95, MPA05], Mazurkiewicz traces [Maz86, GK10], parallel series [LW01], Message Sequence Charts graphs [MR04], and Petri Nets [NPW81]
• An execution of a parallel program is best seen as a **partial order** (happens-before)\(^1\).

• Typical RV formalisms\(^2\) operate on a **total order** (sequence) of events.

---

\(^1\) Consistent with weak memory consistency models [AG96, ANB\(^+\)95, MPA05], Mazurkiewicz traces [Maz86, GK10], parallel series [LW01], Message Sequence Charts graphs [MR04], and Petri Nets [NPW81]

\(^2\) LTL, MTL [TR05], CFG, ERE, QEA [RCR15], DATE [CPS09], LTL\(_3\) monitors [BLS11].
• An execution of a parallel program is best seen as a **partial order** (happens-before)\(^1\).

• Typical RV formalisms\(^2\) operate on a **total order** (sequence) of events.

---

\(^1\) Consistent with weak memory consistency models [AG96, ANB⁹⁵, MPA05], Mazurkiewicz traces [Maz86, GK10], parallel series [LW01], Message Sequence Charts graphs [MR04], and Petri Nets [NPW81]

\(^2\) LTL, MTL [TR05], CFG, ERE, QEA [RCR15], DATE [CPS09], LTL\(_3\) monitors [BLS11].
• An execution of a parallel program is best seen as a partial order (happens-before)\(^1\).

• Typical RV formalisms\(^2\) operate on a total order (sequence) of events.

★ An instrumented program must capture the order of events as it happens during the execution to pass it to monitors.

\(^1\) Consistent with weak memory consistency models [AG96, ANB\(^\dagger\)95, MPA05], Mazurkiewicz traces [Maz86, GK10], parallel series [LW01], Message Sequence Charts graphs [MR04], and Petri Nets [NPW81]

\(^2\) LTL, MTL [TR05], CFG, ERE, QEA [RCR15], DATE [CPS09], LTL\(^3\) monitors [BLS11].
MONITORING MULTITHREADED PROGRAMS

- In this tutorial we focus on:
  1. Available tools capable of performing RV for multithreaded programs
  2. Questions to identify the various situations and the appropriate tools for them
MONITORING MULTITHREADED PROGRAMS

• In this tutorial we focus on:
  1. Available tools capable of performing RV for multithreaded programs
  2. Questions to identify the various situations and the appropriate tools for them

⚠ Some tools allow for writing arbitrary monitors, while handling instrumentation.
  1. We lose the ability to generate monitors automatically
  2. Manual monitors can miss information needed for managing concurrency (instrumentation issues)
  3. The process is complicated due to concurrency, and is error-prone (we show it later)
MONITORING MULTITHREADED PROGRAMS

• In this tutorial we focus on:
  1. Available tools capable of performing RV for multithreaded programs
  2. Questions to identify the various situations and the appropriate tools for them

⚠ Some tools allow for writing arbitrary monitors, while handling instrumentation.
  1. We lose the ability to generate monitors automatically
  2. Manual monitors can miss information needed for managing concurrency (instrumentation issues)
  3. The process is complicated due to concurrency, and is error-prone (we show it later)

★ (Q0) “Is the developer using the tool to automatically generate monitor logic?”

In this tutorial, we only concern ourselves with the tools that do so.
RV & Multithreaded Programs

Approaches Verifying Sequences of Events

Approaches Focusing on Concurrency Errors

Approaches Utilizing Multiple Traces

Conclusion
Approaches Verifying Sequences of Events
## Linear Specifications for Concurrent Programs

<table>
<thead>
<tr>
<th>Tool</th>
<th>Formalisms</th>
<th>Instrumentation Support</th>
</tr>
</thead>
<tbody>
<tr>
<td>JAVA-MOP [CR05]</td>
<td>LTL, FSMs, (P)LTL, CFG, ERE, SRS</td>
<td>✓ (AspectJ)</td>
</tr>
</tbody>
</table>

**STARs**

- (Q1) "Are the models of the specification formalism based on a total order?"

**Linearize Concurrency so that the Trace is a Sequence.**

1. Treat each thread independently (Perthread monitoring)
   - Use flags (perthread) or slicing (∀ \( t \in \text{threads} \))
2. Lock the monitor and linearize its input (Global monitoring)

**Alternatively:**

- Write monitors manually + unsynchronized access to monitor
## Linear Specifications for Concurrent Programs

<table>
<thead>
<tr>
<th>Tool</th>
<th>Formalisms</th>
<th>Instrumentation Support</th>
</tr>
</thead>
<tbody>
<tr>
<td>JAVA-MOP [CR05]</td>
<td>LTL, FSMs, (P)LTL, CFG, ERE, SRS</td>
<td>✓ (AspectJ)</td>
</tr>
<tr>
<td>TRACEMATCHES [BHL+10]</td>
<td>Regular Expressions</td>
<td>✓ (AspectJ)</td>
</tr>
</tbody>
</table>

### Checking Sequences

- Instrumentation Tool Formalisms Support
  - JAVA-MOP [CR05] LTL, FSMs, (P)LTL, CFG, ERE, SRS ✓ (AspectJ)
  - TRACEMATCHES [BHL+10] Regular Expressions ✓ (AspectJ)

### Checking Concurrency

- Multi-Trace
  - Java-MOP
  - Tool
  - Formalisms
  - Instrumentation
  - Support

### Conclusion

- Linear specifications for concurrent programs
- Instrumentation
- Tool
- Formalisms
- Support

- ⋆ (Q1) “Are the models of the specification formalism based on a total order?”
- Linearize concurrency so that the trace is a sequence.

1. Treat each thread independently (Perthread monitoring)
   - Use flags (perthread) or slicing ( ∀ t \in threads)
2. Lock the monitor and linearize its input (Global monitoring)

- Alternatively:
  - Write monitors manually + unsynchronized access to monitor
## Linear Specifications for Concurrent Programs

<table>
<thead>
<tr>
<th>Tool</th>
<th>Formalisms</th>
<th>Instrumentation Support</th>
</tr>
</thead>
<tbody>
<tr>
<td>JAVA-MOP [CR05]</td>
<td>LTL, FSMs, (P)LTL, CFG, ERE, SRS</td>
<td>✓ (AspectJ)</td>
</tr>
<tr>
<td>TRACEMATCHES [BHL+10]</td>
<td>Regular Expressions</td>
<td>✓ (AspectJ)</td>
</tr>
<tr>
<td>MARQ [RCR15]</td>
<td>QEA (Automata-based, using custom DSL)</td>
<td>✗</td>
</tr>
</tbody>
</table>

*For these tools, the trace is expected to be a sequence.*

- **Linearize concurrency so that the trace is a sequence.**
  1. Treat each thread independently (Perthread monitoring)
    - Use flags (perthread) or slicing (\(\forall t \in \text{threads}\))
  2. Lock the monitor and linearize its input (Global monitoring)

- Alternatively: Write monitors manually + unsynchronized access to monitor
## LINEAR SPECIFICATIONS FOR CONCURRENT PROGRAMS

<table>
<thead>
<tr>
<th>Tool</th>
<th>Formalisms</th>
<th>Instrumentation Support</th>
</tr>
</thead>
<tbody>
<tr>
<td>JAVA-MOP [CR05]</td>
<td>LTL, FSMs, (P)LTL, CFG, ERE, SRS</td>
<td>✓ (AspectJ)</td>
</tr>
<tr>
<td>TRACEMATCHES [BHL+10]</td>
<td>Regular Expressions</td>
<td>✓ (AspectJ)</td>
</tr>
<tr>
<td>MARQ [RCR15]</td>
<td>QEA (Automata-based, using custom DSL)</td>
<td>✗</td>
</tr>
<tr>
<td>LARVA [CPS09]</td>
<td>DATE (Automata-based)</td>
<td>✓ (AspectJ)</td>
</tr>
</tbody>
</table>
For these tools, the trace is expected to be a sequence of events.

<table>
<thead>
<tr>
<th>Tool</th>
<th>Formalisms</th>
<th>Instrumentation Support</th>
</tr>
</thead>
<tbody>
<tr>
<td>JAVA-MOP [CR05]</td>
<td>LTL, FSMs, (P)LTL, CFG, ERE, SRS</td>
<td>✓ (AspectJ)</td>
</tr>
<tr>
<td>TRACEMATCHES [BHL+10]</td>
<td>Regular Expressions</td>
<td>✓ (AspectJ)</td>
</tr>
<tr>
<td>MARQ [RCR15]</td>
<td>QEA (Automata-based, using custom DSL)</td>
<td>X</td>
</tr>
<tr>
<td>LARVA [CPS09]</td>
<td>DATE (Automata-based)</td>
<td>✓ (AspectJ)</td>
</tr>
</tbody>
</table>
**LINEAR SPECIFICATIONS FOR CONCURRENT PROGRAMS**

<table>
<thead>
<tr>
<th>Tool</th>
<th>Formalisms</th>
<th>Instrumentation Support</th>
</tr>
</thead>
<tbody>
<tr>
<td>JAVA-MOP [CR05]</td>
<td>LTL, FSMs, (P)LTL, CFG, ERE, SRS</td>
<td>✓ (AspectJ)</td>
</tr>
<tr>
<td>TRACEMATCHES [BHL+10]</td>
<td>Regular Expressions</td>
<td>✓ (AspectJ)</td>
</tr>
<tr>
<td>MARQ [RCR15]</td>
<td>QEA (Automata-based, using custom DSL)</td>
<td>✗</td>
</tr>
<tr>
<td>LARVA [CPS09]</td>
<td>DATE (Automata-based)</td>
<td>✓ (AspectJ)</td>
</tr>
</tbody>
</table>

- For these tools, the trace is expected to be a **sequence** of events.

★ (Q1) “Are the models of the specification formalism based on a total order?”
LINEAR SPECIFICATIONS FOR CONCURRENT PROGRAMS

<table>
<thead>
<tr>
<th>Tool</th>
<th>Formalisms</th>
<th>Instrumentation Support</th>
</tr>
</thead>
<tbody>
<tr>
<td>JAVA-MOP [CR05]</td>
<td>LTL, FSMs, (P)LTL, CFG, ERE, SRS</td>
<td>✓ (AspectJ)</td>
</tr>
<tr>
<td>TRACEMATCHES [BHL+10]</td>
<td>Regular Expressions</td>
<td>✓ (AspectJ)</td>
</tr>
<tr>
<td>MARQ [RCR15]</td>
<td>QEA (Automata-based, using custom DSL)</td>
<td>X</td>
</tr>
<tr>
<td>LARVA [CPS09]</td>
<td>DATE (Automata-based)</td>
<td>✓ (AspectJ)</td>
</tr>
</tbody>
</table>

- For these tools, the trace is expected to be a **sequence** of events.
- *(Q1) “Are the models of the specification formalism based on a total order?”*
- **Linearize** concurrency so that the trace is a sequence.
LINEAR SPECIFICATIONS FOR CONCURRENT PROGRAMS

<table>
<thead>
<tr>
<th>Tool</th>
<th>Formalisms</th>
<th>Instrumentation Support</th>
</tr>
</thead>
<tbody>
<tr>
<td>JAVA-MOP [CR05]</td>
<td>LTL, FSMs, (P)LTL, CFG, ERE, SRS</td>
<td>✓ (AspectJ)</td>
</tr>
<tr>
<td>TRACEMATCHES [BHL+10]</td>
<td>Regular Expressions</td>
<td>✓ (AspectJ)</td>
</tr>
<tr>
<td>MARQ [RCR15]</td>
<td>QEA (Automata-based, using custom DSL)</td>
<td>✗</td>
</tr>
<tr>
<td>LARVA [CPS09]</td>
<td>DATE (Automata-based)</td>
<td>✓ (AspectJ)</td>
</tr>
</tbody>
</table>

- For these tools, the trace is expected to be a sequence of events.

★ (Q1) “Are the models of the specification formalism based on a total order?”

- Linearize concurrency so that the trace is a sequence.
  1. Treat each thread independently (Perthread monitoring)
     Use flags (perthread) or slicing (∀t ∈ threads)
LINEAR SPECIFICATIONS FOR CONCURRENT PROGRAMS

<table>
<thead>
<tr>
<th>Tool</th>
<th>Formalisms</th>
<th>Instrumentation Support</th>
</tr>
</thead>
<tbody>
<tr>
<td>JAVA-MOP [CR05]</td>
<td>LTL, FSMs, (P)LTL, CFG, ERE, SRS</td>
<td>✓ (AspectJ)</td>
</tr>
<tr>
<td>TRACEMATCHES [BHL+10]</td>
<td>Regular Expressions</td>
<td>✓ (AspectJ)</td>
</tr>
<tr>
<td>MARQ [RCR15]</td>
<td>QEA (Automata-based, using custom DSL)</td>
<td>X</td>
</tr>
<tr>
<td>LARVA [CPS09]</td>
<td>DATE (Automata-based)</td>
<td>✓ (AspectJ)</td>
</tr>
</tbody>
</table>

- For these tools, the trace is expected to be a sequence of events.
- *(Q1) “Are the models of the specification formalism based on a total order?”*
- Linearize concurrency so that the trace is a sequence.
  1. Treat each thread *independently* (Perthread monitoring)
     Use flags (perthread) or slicing ($\forall t \in$ threads)
  2. Lock the monitor and linearize its input (Global monitoring)
LINEAR SPECIFICATIONS FOR CONCURRENT PROGRAMS

<table>
<thead>
<tr>
<th>Tool</th>
<th>Formalisms</th>
<th>Instrumentation Support</th>
</tr>
</thead>
<tbody>
<tr>
<td>JAVA-MOP [CR05]</td>
<td>LTL, FSMs, (P)LTL, CFG, ERE, SRS</td>
<td>✓ (AspectJ)</td>
</tr>
<tr>
<td>TRACEMATCHES [BHL+10]</td>
<td>Regular Expressions</td>
<td>✓ (AspectJ)</td>
</tr>
<tr>
<td>MARQ [RCR15]</td>
<td>QEA (Automata-based, using custom DSL)</td>
<td>X</td>
</tr>
<tr>
<td>LARVA [CPS09]</td>
<td>DATE (Automata-based)</td>
<td>✓ (AspectJ)</td>
</tr>
</tbody>
</table>

- For these tools, the trace is expected to be a sequence of events.
- (Q1) “Are the models of the specification formalism based on a total order?”
- Linearize concurrency so that the trace is a sequence.
  1. Treat each thread independently (Perthread monitoring)
     Use flags (perthread) or slicing (∀t ∈ threads)
  2. Lock the monitor and linearize its input (Global monitoring)
- Alternatively:
  Write monitors manually + unsynchronized access to monitor

A. El-Hokayem, Y. Falcone, Can We Monitor All Multithreaded Programs?
MONITORING A SIMPLE PROGRAM

- Example: Given a linked list, each thread processes the list independently
  - Compute avg, min, max etc.
- (SafeIter) For an iterator: always call `hasNext` before calling `next`
MONITORING A SIMPLE PROGRAM

- Example: Given a linked list, each thread processes the list independently
  - Compute avg, min, max etc.
- (SafeIter) For an iterator: always call `hasNext` before calling `next`

- Example found in `scenarios/process`
- Let us begin by using a single monitor to check the property
- Follow the tutorial until reaching the end of “Developing The Global Monitor (Simplest)”
PERTHREAD MONITORING

- Monitor each thread for a given property *independently* of other threads.
- Java-MOP/Tracematches (perthread flag), LARVA/MarQ (slice on thread)

⋆ We need to check the properties \( \phi_1 \) and \( \phi_2 \) across threads

⋆ (Q3) “Does there exist a model of the specification where events are generated by more than a single thread?”
## Perthread Monitoring

- Monitor each thread for a given property *independently* of other threads.
- Java-MOP/Tracematches (perthread flag), LARVA/MarQ (slice on thread)

- Continue tutorial in *scenarios/process*
PERTHREAD MONITORING

- Monitor each thread for a given property independently of other threads.
- Java-MOP/Tracematches (perthread flag), LARVA/MarQ (slice on thread)

- Continue tutorial in scenarios/process

⚠️ Can we use perthread monitoring to monitor producer-consumer?
PER THREAD MONITORING

• Monitor each thread for a given property independently of other threads.
• Java-MOP/Tracematches (perthread flag), LARVA/MarQ (slice on thread)

• Continue tutorial in scenarios/process

⚠️ Can we use perthread monitoring to monitor producer-consumer?
  • Produce and consume events are observed in separate threads
    ★ We need to check the properties (\(\varphi_1\) and \(\varphi_2\)) across threads
PER THREAD MONITORING

- Monitor each thread for a given property independently of other threads.
- Java-MOP/Tracematches (perthread flag), LARVA/MarQ (slice on thread)

- Continue tutorial in scenarios/process

⚠️ Can we use perthread monitoring to monitor producer-consumer?
  - Produce and consume events are observed in separate threads
    ⭐ We need to check the properties ($\varphi_1$ and $\varphi_2$) across threads

⭐ (Q3) “Does there exist a model of the specification where events are generated by more than a single thread?”
GLOBAL MONITORING

- **Lock** monitor to feed it with events across *multiple* threads.
GLOBAL MONITORING

- **Lock** monitor to feed it with events across *multiple* threads.

- Let us monitor *producer-consumer*. 
GLOBAL MONITORING

- **Lock** monitor to feed it with events across multiple threads.

- Let us monitor producer-consumer.

- Follow tutorial in scenarios/producer-consumer-v1
GLOBAL MONITORING

- **Lock** monitor to feed it with events across **multiple** threads.

- Let us monitor **producer-consumer**.

- Follow tutorial in **scenarios/producer-consumer-v1**

- Notice how everything runs smoothly
GLOBAL MONITORING

- **Lock** monitor to feed it with events across *multiple* threads.

- Let us monitor **producer-consumer**.

- Follow tutorial in **scenarios/producer-consumer-v1**
  - Notice how everything runs smoothly
  - That is because this variant (variant 1) is **correct**.
- The usage of **locks** is ensures that a correct trace is generated.
GLOBAL MONITORING

- **Lock** monitor to feed it with events across **multiple** threads.

- Let us monitor **producer-consumer**.

- Follow tutorial in **scenarios/producer-consumer-v1**
  - Notice how everything runs smoothly
  - That is because this variant (variant 1) is **correct**.
  - The usage of **locks** is ensures that a correct trace is generated.

- Now let us monitor a non-synchronized **producer-consumer**.
  - Follow tutorial in **scenarios/producer-consumer-v2**
GLOBAL MONITORING

- **Lock** monitor to feed it with events across *multiple* threads.
- Let us monitor *producer-consumer*.
- Follow tutorial in *scenarios/producer-consumer-v1*
  - Notice how everything runs smoothly
  - That is because this variant (variant 1) is **correct**.
  - The usage of **locks** is ensures that a correct trace is generated.
- Now let us monitor a non-synchronized *producer-consumer*.
  - Follow tutorial in *scenarios/producer-consumer-v2*

⚠️ You will notice different verdicts reported for different runs
• With the absence of locks, events can happen **concurrently**.
  • In Variant 1: locks guarantee a **sequence**
  • In Variant 2: no locks, monitors will **linearize** arbitrarily
GLOBAL MONITORING \(\Leftrightarrow\) (Cont’d)

Thread 0

\[
\begin{align*}
1 & \rightarrow 2 \\
3 & \rightarrow 4 \\
5 & \rightarrow 6
\end{align*}
\]

Thread 1

\[
\begin{align*}
1 & \rightarrow 3 \\
2 & \rightarrow 5 \\
4 & \rightarrow 6
\end{align*}
\]

\begin{itemize}
  \item With the absence of locks, events can happen \textit{concurrently}.
  \item In Variant 1: locks guarantee a sequence
  \item In Variant 2: no locks, monitors will \textit{linearize} arbitrarily
\end{itemize}

⚠️ Does it suffice to simply use \textit{locks} on the monitor?
### Differences in Variants and Tools

<table>
<thead>
<tr>
<th>V</th>
<th>Consumers</th>
<th>Tool</th>
<th>Advice</th>
<th>True</th>
<th>False</th>
<th>Timeout</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1-2</td>
<td>REF</td>
<td>-</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>JMOP</td>
<td>10,000</td>
<td>(100%)</td>
<td>0</td>
<td>(0%)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>10,000</td>
<td>(100%)</td>
<td>0</td>
<td>(0%)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>MarQ</td>
<td>10,000</td>
<td>(100%)</td>
<td>0</td>
<td>(0%)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>10,000</td>
<td>(100%)</td>
<td>0</td>
<td>(0%)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>LARVA</td>
<td>A</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>B</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>REF</td>
<td></td>
<td>631</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>JMOP</td>
<td>4,043</td>
<td>(40.43%)</td>
<td>5,957</td>
<td>(59.57%)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>7,175</td>
<td>(71.75%)</td>
<td>6</td>
<td>(0.06%)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>MarQ</td>
<td>4,404</td>
<td>(44.04%)</td>
<td>5,583</td>
<td>(55.83%)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>9,973</td>
<td>(99.73%)</td>
<td>16</td>
<td>(0.16%)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>LARVA</td>
<td>A</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>B</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>REF</td>
<td></td>
<td>4,785</td>
<td>4,785</td>
<td>(47.85%)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>JMOP</td>
<td>128</td>
<td>(1.28%)</td>
<td>9,220</td>
<td>(92.20%)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1,260</td>
<td>(12.60%)</td>
<td>7,617</td>
<td>(76.17%)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>MarQ</td>
<td>33</td>
<td>(0.33%)</td>
<td>9,957</td>
<td>(99.57%)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>432</td>
<td>(4.32%)</td>
<td>9,530</td>
<td>(95.30%)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>LARVA</td>
<td>A</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>B</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Verifying 10,000 executions of the two variants of producer-consumer, using before/after instrumentation points with respect to \( \varphi_1 \) and \( \varphi_2 \).

A. El-Hokayem, Y. Falcone, Can We Monitor All Multithreaded Programs?
INSTRUMENTING CONCURRENT PROGRAMS

- Programs are typically instrumented to generate events.
- Locking a monitor guarantees that events are processed as a sequence.
INSTRUMENTING CONCURRENT PROGRAMS

- Programs are typically instrumented to generate events.
- Locking a monitor guarantees that events are processed as a sequence.

★ But is the order of the captured events the same as that which happened during the execution?
INSTRUMENTING CONCURRENT PROGRAMS

- Programs are typically instrumented to generate events.
- Locking a monitor guarantees that events are processed as a sequence.

★ But is the order of the captured events the same as that which happened during the execution?

- The code needed to call the monitor usually is not atomic with the event occurring.
INSTRUMENTING CONCURRENT PROGRAMS

- Programs are typically instrumented to generate events.
- Locking a monitor guarantees that events are processed as a sequence.

★ But is the order of the captured events the same as that which happened during the execution?

- The code needed to call the monitor usually is not atomic with the event occurring.

→ A context switch may happen in between

Thread 0

\[ f() \]

\[ \text{mon}(f) \]

Thread 1

\[ g() \]

\[ \text{mon}(g) \]
Programs are typically instrumented to generate events.
Locking a monitor guarantees that events are processed as a sequence.

But is the order of the captured events the same as that which happened during the execution?

The code needed to call the monitor usually is not atomic with the event occurring.

A context switch may happen in between

\[ \text{Thread 0: } f() \rightarrow \text{mon}(f) \]
\[ \text{Thread 1: } g() \rightarrow \text{mon}(g) \]

The trace does not represent the actual order of events in the execution.
INSTRUMENTATION

- To check for this behavior we will design a simple logging program.
- We have two functions \( f() \) and \( g() \) called by separate threads multiple times.
- The functions print \( f \) and \( g \), respectively.
- We instrument before/after them to call a monitor which prints \( f\_trace \) and \( g\_trace \), respectively.
- We compare the order of the events in the trace and actual calls.
INSTRUMENTATION

- To check for this behavior we will design a simple logging program.
- We have two functions f() and g() called by separate threads multiple times.
- The functions print f and g, respectively.
- We instrument before/after them to call a monitor which prints f_trace and g_trace, respectively.
- We compare the order of the events in the trace and actual calls.

- Follow the tutorial in scenarios/collect
## Instrumentation ➫ Results

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>g</td>
<td>f_trace</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>f</td>
<td>g_trace</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>f</td>
<td>f_trace</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>g</td>
<td>g_trace</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>f</td>
<td>f_trace</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>g</td>
<td>g_trace</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>f</td>
<td>f_trace</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>g</td>
<td>g_trace</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>f</td>
<td>f_trace</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>g</td>
<td>f_trace</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>f</td>
<td>g_trace</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>f</td>
<td>f_trace</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>g</td>
<td>g_trace</td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>g</td>
<td>g_trace</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Tool</th>
<th>Advice</th>
<th>Sync</th>
<th>Identical</th>
<th>Different</th>
</tr>
</thead>
<tbody>
<tr>
<td>AspectJ</td>
<td>A</td>
<td>✓</td>
<td>4,912</td>
<td>5,088</td>
</tr>
<tr>
<td></td>
<td>B</td>
<td></td>
<td>9,170</td>
<td>830</td>
</tr>
<tr>
<td>Java-MOP</td>
<td>A</td>
<td>✓</td>
<td>1,737</td>
<td>8,263</td>
</tr>
<tr>
<td></td>
<td>B</td>
<td></td>
<td>9,749</td>
<td>251</td>
</tr>
<tr>
<td>LARVA</td>
<td>A</td>
<td>✓</td>
<td>8,545</td>
<td>1,455</td>
</tr>
<tr>
<td></td>
<td>B</td>
<td></td>
<td>9,992</td>
<td>8</td>
</tr>
<tr>
<td>Java-MOP</td>
<td>A</td>
<td>x</td>
<td>2,026</td>
<td>7,974</td>
</tr>
<tr>
<td></td>
<td>B</td>
<td></td>
<td>9,517</td>
<td>483</td>
</tr>
</tbody>
</table>
LINEAR SPECIFICATIONS $\leftrightarrow$ GOOD SITUATIONS

- Basic idea: check sequences when the events are indeed found as a sequence in the program
LINEAR SPECIFICATIONS → GOOD SITUATIONS

- Basic idea: check sequences when the events are indeed found as a sequence in the program

- Is it sufficient to simply ensure the program is correctly synchronized?
LINEAR SPECIFICATIONS  \(\rightarrow\) GOOD SITUATIONS

- Basic idea: check sequences when the events are indeed found as a sequence in the program

- Is it sufficient to simply ensure the program is correctly synchronized?
- No, the program can still have concurrent events regardless (ex: list processing / readers-writers)
**LINEAR SPECIFICATIONS → GOOD SITUATIONS**

- Basic idea: check sequences when the events are indeed found as a sequence in the program

- Is it sufficient to simply ensure the program is **correctly synchronized**?
- **No**, the program can still have **concurrent events** regardless (ex: list processing / readers-writers)

★ (Q4) “Is the satisfaction of the specification sensitive to the order of concurrent events?”
Approaches Focusing on Concurrency Errors
### VERIFYING CONCURRENCY CORRECTNESS

<table>
<thead>
<tr>
<th>Tool</th>
<th>Properties</th>
<th>Theroetical Model</th>
<th>Online</th>
</tr>
</thead>
<tbody>
<tr>
<td>JPAX [HR04]</td>
<td>DRF/DF + LTL</td>
<td>Lockset-based/ERASER [SBN+97], sequential consistency only</td>
<td>✔️</td>
</tr>
</tbody>
</table>

These tools verify specific hard-coded (“low-level”) concurrency properties.

- **DRF**: data race freedom
- **DF**: deadlock freedom

General behavior properties are not always checked. GPredict allows for behavioral properties but offline.
## VERIFYING CONCURRENCY CORRECTNESS

<table>
<thead>
<tr>
<th>Tool</th>
<th>Properties</th>
<th>Theroetical Model</th>
<th>Online</th>
</tr>
</thead>
<tbody>
<tr>
<td>JPAX [HR04]</td>
<td>DRF/DF + LTL</td>
<td>Lockset-based/ERASER [SBN+97], sequential consistency only</td>
<td>✓</td>
</tr>
<tr>
<td>RVPredict [HMR14]</td>
<td>DRF</td>
<td>PTA-based</td>
<td>✗</td>
</tr>
<tr>
<td>GPredict [HLR15]</td>
<td>DRF + RE</td>
<td>Maximal Causal Model [HMR14]</td>
<td>✗</td>
</tr>
<tr>
<td></td>
<td>Atomic regions Concurrency</td>
<td>PTA-based Maximal Causal Model [HMR14]</td>
<td></td>
</tr>
</tbody>
</table>
### VERIFYING CONCURRENCY CORRECTNESS

<table>
<thead>
<tr>
<th>Tool</th>
<th>Properties</th>
<th>Theroetical Model</th>
<th>Online</th>
</tr>
</thead>
<tbody>
<tr>
<td>JPAX [HR04]</td>
<td>DRF/DF + LTL</td>
<td>Lockset-based/ERASER [SBN+97], sequential consistency only</td>
<td>✓</td>
</tr>
<tr>
<td>RVPredict [HMR14]</td>
<td>DRF</td>
<td>PTA-based</td>
<td>✗</td>
</tr>
<tr>
<td>GPredict [HLR15]</td>
<td>DRF + RE Atomic regions Concurrency</td>
<td>Maximal Causal Model [HMR14]</td>
<td>✗</td>
</tr>
</tbody>
</table>

- These tools verify specific hard-coded (“low-level”) **Concurrency** properties.
  - DRF: data race freedom
  - DF: deadlock freedom
- General behavior properties are **not always checked**.
- **GPredict** allows for behavioral properties but **offline**

---

A. El-Hokayem, Y. Falcone, Can We Monitor All Multithreaded Programs?

17
Gpredict specification (from [HLR15])

Gpredict specification (from [HLR15])

```cpp
AtomicityViolation (Object o) {
  event begin before (Object o) : execution(m());
  event read before (Object o) : get(*s) && target(o);
  event write before (Object o) : set(*s) && target(o);
  event end after (Object o) : execution(m());

  pattern: begin(t1, <r1)
    read(t1) write(t2) write(t1)
    end(t1, >r1)

  pattern: read(t1) || write(t2)
```

A. El-Hokayem, Y. Falcone, Can We Monitor All Multithreaded Programs?
Approaches Utilizing Multiple Traces
Tools that utilize multiple traces as input (or it can be seen that way)

Techniques/tools need adaptation for multithreaded context
Tools that utilize multiple traces as input (or it can be seen that way) → Techniques/tools need adaptation for multithreaded context

1. Stream-based Runtime Verification:
   - Utilizes operations (arbitrary functions) that aggregate streams (of events): timing/delays, filters, and statistical
   - Tools/specification languages:
     LOLA \([DSS^+05]\), TeSSLa \([LSS^+18, CHL^+18]\), BEEPBEEP \([HK17]\)
**Multi-trace RV ↔ Stream-Based RV**

★ Tools that utilize *multiple traces* as input (or it can be seen that way)

→ Techniques/tools need *adaptation* for multithreaded context

1. **Stream-based Runtime Verification:**
   - Utilizes operations (arbitrary functions) that *aggregate* streams (of events): timing/delays, filters, and statistical
   - Tools/specification languages:
     LOLA [DSS⁺05], TeSSLa [LSS⁺18, CHL⁺18], BEEPBEEP [HK17]

→ Streams for each thread, and determine aggregation function in a multithreaded context.
2. Decentralized monitoring/specifications

- Monitoring over multiple components, each having its own trace
- Tools: DecentMon [BF16, CF16], THEMIS [EF17b]
- Automata-based [FCF14, EF17a], distributed systems (predicate detection [NCMG17, BFRT16], (pt)DTL [SS14])
2. **Decentralized monitoring/specifications**

   - Monitoring over *multiple components*, each having its own trace
   - Tools: DecentMon [BF16, CF16], THEMIS [EF17b]
   - Automata-based [FCF14, EF17a], distributed systems (predicate detection [NCMG17, BFRT16], (pt)DTL [SS14])

   → Each thread is seen as a component, communication between components need to be efficient in a multithreaded context
3. Hyperproperties [FRS15]
   - Consider multiple traces of the same program (possibly different executions)
   - Used for verifying security policies
   - Tool: RVHyper [FHST18]
3. Hyperproperties [FRS15]

- Consider multiple traces of the same program (possibly different executions)
- Used for verifying security policies
- Tool: RVHyper [FHST18]

→ Multiple traces → express multiple possible re-orderings → concurrency as a hyperproperty
Conclusion
CONCLUSION

- Requires monitor synthesis (Q0)?
  - Yes
  - No

- Using total-order formalism (Q1)?
  - Yes
  - No

- Monitoring specific properties (Q2)?
  - Yes
  - No

  - Multi-Trace
  - Stream-Based RV
  - Decentralized Monitoring
  - Decentralized Specifications
  - RV of Hyperproperties
  - Need Adaptation/Tools

- Events across threads (Q3)?
  - Yes
  - No

  - Data race/Deadlock
    - RVPredict
    - JPaX

  - Per-thread
    - Java-MOP
    - Tracematches
    - MarQ

- Events in concurrent regions (Q4)?
  - Yes
  - No

  - Global
    - Java-MOP
    - Tracematches
    - MarQ
    - LARVA

⚠️: Non-determinism and trace collection issues.


References III


