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

Instructions ↪ Prepare the tutorial files

1. The tutorial repository is hosted at
   `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:

```
1 —— rv-multithreaded ——
2
3  root's password is 'root'
4
5 ——————————
6    Browse the README files:
7    http://localhost:8050/
8 ——————————
9
10 [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

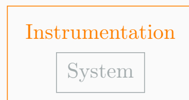
Specification

System

Context ↪ Brief Overview of RV

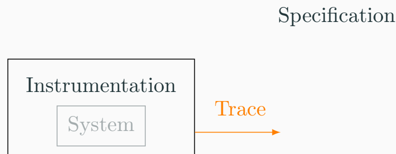
- Lightweight verification technique
- Checks whether a run of a program conforms to a specification
  (As opposed to model checking which verifies all runs)
- The run is captured as a trace, typically seen as a sequence of events

Specification



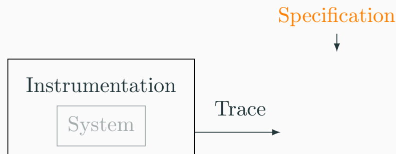
Context ↪ Brief Overview of RV

- 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



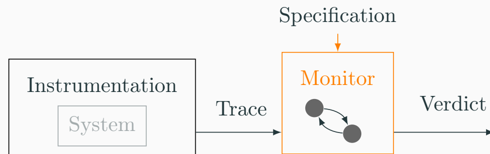
Context ↪ Brief Overview of RV

- 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



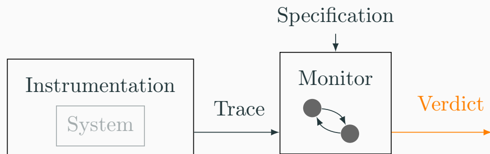
Context ↪ Brief Overview of RV

- 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
- 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)



Context ↪ 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

Context ↪ 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. (φ_1) Consumers must not remove an item unless the queue contains one
  2. (φ_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 }
```

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 }
```

Thread 0 (Producer)	Thread 1 (Consumer)
① `sq.produce(0);`	② `sq.consume(); //0`
③ `sq.produce(1);`	④ `sq.consume(); //1`

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 }
```

Thread 0 (Producer)	Thread 1 (Consumer)
① `sq.produce(0);`	② `sq.consume(); //0`
③ `sq.produce(1);`	④ `sq.consume(); //1`

Execution	Verdict
①②③④	$\top$

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 }
```

Thread 0 (Producer)	Thread 1 (Consumer)
① `sq.produce(0);`	② `sq.consume(); //0`
③ `sq.produce(1);`	④ `sq.consume(); //1`

Execution	Verdict
①②③④	$\top$
①③②④	$\top$

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 }
```

Thread 0 (Producer)	Thread 1 (Consumer)
① `sq.produce(0);`	② `sq.consume(); //0`
③ `sq.produce(1);`	④ `sq.consume(); //1`

Execution	Verdict	
1 2 3 4	$\top$	
1 3 2 4	$\top$	
2 1 3 4	$\bot$	Consume on an empty queue (φ_1)

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 }
```

Thread 0 (Producer)	Thread 1 (Consumer)
① `sq.produce(0);`	② `sq.consume(); //0`
③ `sq.produce(1);`	④ `sq.consume(); //1`

Execution	Verdict	
1 2 3 4	$\top$	
1 3 2 4	$\top$	
2 1 3 4	$\bot$	Consume on an empty queue (φ_1)
1 3 2	$\bot$	One element left in queue (φ_2)

Context ↪ Executions of Producer-consumer

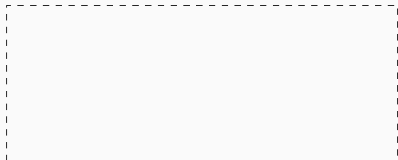
```
1 public class SynchQueue {
2   private LinkedList<Integer> q = new LinkedList<Integer>();
3   public void    produce(Integer v)    { q.add(v); }
4   public Integer consume()       { return q.poll(); }
5 }
```

Thread 0 (Producer)	Thread 1 (Consumer)
① `sq.produce(0);`	② `sq.consume(); //0`
③ `sq.produce(1);`	④ `sq.consume(); //1`

Execution	Verdict	
①②③④	$\top$	
①③②④	$\top$	
②①③④	$\bot$	Consume on an empty queue (φ_1)
①③②	$\bot$	One element left in queue (φ_2)
②④①③	$\bot$	Violates both (φ_1) and (φ_2)

CHALLENGES

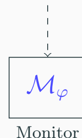
Instrumentation



Trace

Specification (φ)

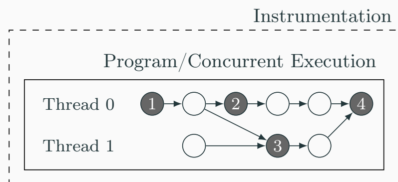
Verdict



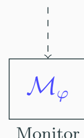
1

2

CHALLENGES



Trace Specification (φ) Verdict

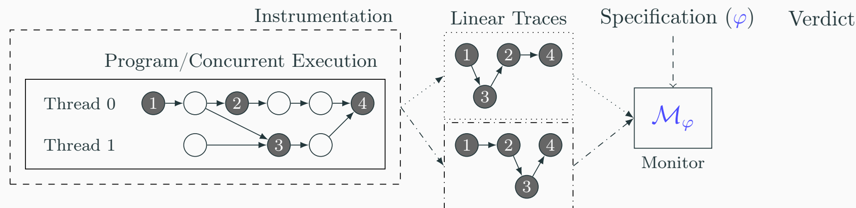


- An execution of a parallel program is best seen as a partial order (happens-before)[1].

[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]

[2]

CHALLENGES

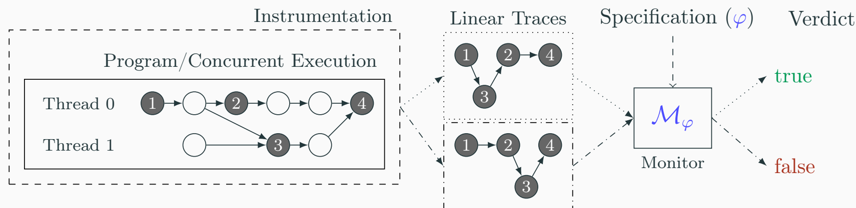


- An execution of a parallel program is best seen as a partial order (happens-before)[1].
- Typical RV formalisms[2] operate on a total order (sequence) of events.

[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].

CHALLENGES

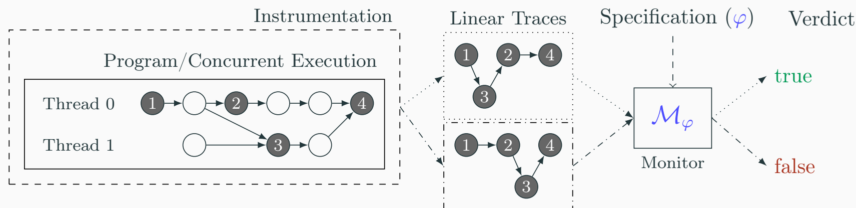


- An execution of a parallel program is best seen as a partial order (happens-before)[1].
- Typical RV formalisms[2] operate on a total order (sequence) of events.

[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].

CHALLENGES



- 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+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

Tool	Formalisms	Instrumentation Support
JAVA-MOP [CR05]	LTL, FSMs, (P)LTL, CFG, ERE, SRS	✓ (AspectJ)

LINEAR SPECIFICATIONS FOR CONCURRENT PROGRAMS

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

LINEAR SPECIFICATIONS FOR CONCURRENT PROGRAMS

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

LINEAR SPECIFICATIONS FOR CONCURRENT PROGRAMS

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

LINEAR SPECIFICATIONS FOR CONCURRENT PROGRAMS

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

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

LINEAR SPECIFICATIONS FOR CONCURRENT PROGRAMS

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

- 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

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

- 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

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

- 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)

LINEAR SPECIFICATIONS FOR CONCURRENT PROGRAMS

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

- 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

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

- 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)
- Alternatively:
  Write monitors manually + unsynchronized access to monitor

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)

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?

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?

  - Produce and consume events are observed in separate threads
  - ⋆ We need to check the properties (φ_1 and φ_2) across threads

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?
  - Produce and consume events are observed in separate threads
  - ⋆ We need to check the properties (φ_1 and φ_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.

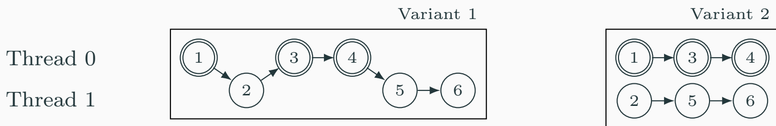
- Let us monitor producer-consumer.

- 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`

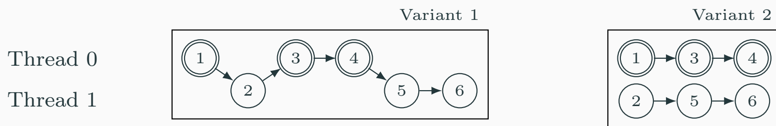
⚠ You will notice different verdicts reported for different runs

Global Monitoring ↪ (Cont'd)



- 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 ↪ (CONT'D)



- 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

⚠ Does it suffice to simply use locks on the monitor?

DIFFERENCES IN VARIANTS AND TOOLS

V	Consumers	Tool	Advice	True #	True %	False #	False %	Timeout #	Timeout %
1	1-2	REF			-			0	(0%)
		JMOP	A	10,000	(100%)	0	(0%)	0	(0%)
			B	10,000	(100%)	0	(0%)	0	(0%)
		MarQ	A	10,000	(100%)	0	(0%)	0	(0%)
			B	10,000	(100%)	0	(0%)	0	(0%)
		LARVA	A	10,000	(100%)	0	(0%)	0	(0%)
			B	10,000	(100%)	0	(0%)	0	(0%)
2	1	REF			-			631	(6.3%)
		JMOP	A	4,043	(40.43%)	5,957	(59.57%)	0	(0%)
			B	7,175	(71.75%)	6	(0.06%)	2,819	(28.19%)
		MarQ	A	4,404	(44.04%)	5,583	(55.83%)	13	(0.13%)
			B	9,973	(99.73%)	16	(0.16%)	11	(0.11%)
		LARVA	A	4,755	(47.55%)	5,245	(52.45%)	0	(0%)
			B	9,988	(99.88%)	2	(0.02%)	10	(0.10%)
2	2	REF			-			4,785	(47.85%)
		JMOP	A	128	(1.28%)	9,220	(92.20%)	652	(6.52%)
			B	1,260	(12.60%)	7,617	(76.17%)	1,123	(11.23%)
		MarQ	A	33	(0.33%)	9,957	(99.57%)	10	(0.10%)
			B	432	(4.32%)	9,530	(95.30%)	38	(0.38%)
		LARVA	A	250	(2.50%)	9,488	(94.88%)	262	(2.62%)
			B	5,823	(58.23%)	4,131	(41.31%)	46	(0.46%)

Verifying 10,000 executions of the two variants of producer-consumer, using before/after instrumentation points with respect to φ_1 and φ_2.

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.

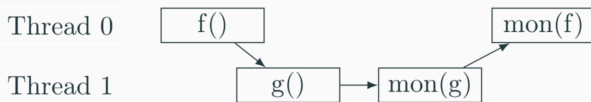
INSTRUMENTING CONCURRENT PROGRAMS

- 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



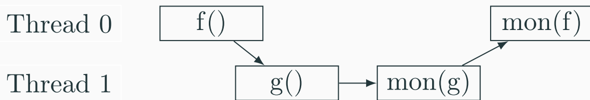
INSTRUMENTING CONCURRENT PROGRAMS

- 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



→ 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

```
 1  g    f_trace
 2  f    g_trace
 3  f    f_trace
 4  g    g_trace
 5  f    f_trace
 6  g    g_trace
 7  f    f_trace
 8  g    g_trace
 9  f    f_trace
10  g    f_trace
11  f    g_trace
12  f    f_trace
13  g    g_trace
14  g    g_trace
```

Tool	Advice	Sync	Identical	Different
AspectJ	A	✓	4,912	5,088
	B		9,170	830
Java-MOP	A	✓	1,737	8,263
	B		9,749	251
LARVA	A	✓	8,545	1,455
	B		9,992	8
Java-MOP	A	✗	2,026	7,974
	B		9,517	483

Linear Specifications ↪ Good situations

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

Linear Specifications ↪ Good situations

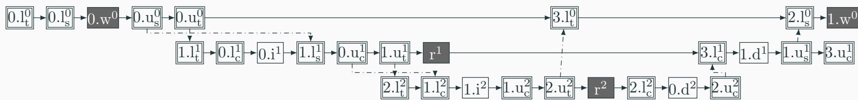
- Basic idea: check sequences when the events are indeed found as a sequence in the program

- Is it sufficient to simply ensure the program is correctly synchronized?

Linear Specifications ↪ Good situations

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

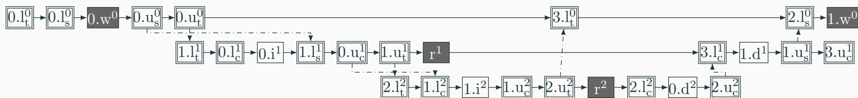
- Is it sufficient to simply ensure the program is correctly synchronized?
- No, the program can still have concurrent events regardless (ex: list processing / readers-writers)



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

Tool	Properties	Theroetical Model	Online
JPAX [HR04]	DRF/DF + LTL	Lockset-based/ERASER [SBN+97], sequential consistency only	✓

VERIFYING CONCURRENCY CORRECTNESS

Tool	Properties	Theroetical Model	Online
JPAX [HR04]	DRF/DF + LTL	Lockset-based/ERASER [SBN+97], sequential consistency only	✓
RVPREDICT [HMR14]	DRF	PTA-based Maximal Causal Model [HMR14]	✗
GPREDICT [HLR15]	DRF + RE Atomic regions Concurrency	PTA-based Maximal Causal Model [HMR14]	✗

VERIFYING CONCURRENCY CORRECTNESS

Tool	Properties	Theroetical Model	Online
JPAX [HR04]	DRF/DF + LTL	Lockset-based/ERASER [SBN+97], sequential consistency only	✓
RVPREDICT [HMR14]	DRF	PTA-based Maximal Causal Model [HMR14]	✗
GPREDICT [HLR15]	DRF + RE Atomic regions Concurrency	PTA-based Maximal Causal Model [HMR14]	✗

- 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

GPREDICT

```
1  AtomicityViolation (Object o){
2    event begin before(Object o) : execution(m());
3    event read  before(Object o) : get(* s) && target(o);
4    event write before(Object o) : set(* s) && target(o);
5    event end   after(Object o)  : execution(m());
6
7    pattern: begin(t1, <r1)
8             read(t1) write(t2) write(t1)
9             end(t1,>r1)
0    pattern: read(t1) || write(t2)
1  }
```

GPredict specification (from [HLR15])

Approaches Utilizing Multiple Traces

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

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]

Multi-trace RV $\hookrightarrow$ 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.

Multi-trace RV ↪ Decentralized Monitoring

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])

Multi-trace RV ↪ Decentralized Monitoring

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

Multi-trace RV ↪ Hyperproperties

3. Hyperproperties [FRS15]
   - Consider multiple traces of the same program (possibly different executions)
   - Used for verifying security policies
   - Tool: RVHyper [FHST18]

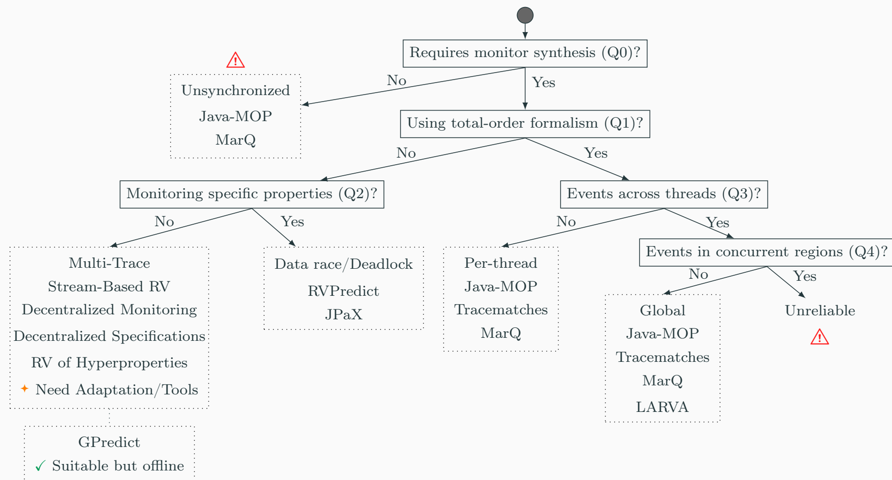
Multi-trace RV ↪ Hyperproperties

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



: Non-determinism and trace collection issues.

References I

S. V. Adve and K. Gharachorloo, *Shared memory consistency models: a tutorial*, Computer **29** (1996), no. 12, 66–76.

Mustaque Ahamad, Gil Neiger, James E. Burns, Prince Kohli, and Phillip W. Hutto, *Causal memory: definitions, implementation, and programming*, Distributed Computing **9** (1995), no. 1, 37–49 (en).

Andreas Bauer and Yliès Falcone, *Decentralised LTL monitoring*, Formal Methods in System Design **48** (2016), no. 1-2, 46–93.

Borzoo Bonakdarpour, Pierre Fraigniaud, Sergio Rajsbaum, and Corentin Travers, *Challenges in fault-tolerant distributed runtime verification*, Leveraging Applications of Formal Methods, Verification and Validation: Discussion, Dissemination, Applications - 7th International Symposium, ISoLA 2016, Imperial, Corfu, Greece, October 10-14, 2016, Proceedings, Part II (Tiziana Margaria and Bernhard Steffen, eds.), Lecture Notes in Computer Science, vol. 9953, 2016, pp. 363–370.

Eric Bodden, Laurie Hendren, Patrick Lam, Ondřej Lhoták, and Nomair A. Naeem, *Collaborative Runtime Verification with Tracematches*, Journal of Logic and Computation **20** (2010), no. 3, 707–723.

Andreas Bauer, Martin Leucker, and Christian Schallhart, *Runtime verification for ltl and tltl*, ACM Trans. Softw. Eng. Methodol. **20** (2011), no. 4, 14:1–14:64.

Tevfik Bultan and Koushik Sen (eds.), *Proceedings of the 26th ACM SIGSOFT international symposium on software testing and analysis, santa barbara, ca, usa, july 10 - 14, 2017*, ACM, 2017.

Christian Colombo and Yliès Falcone, *Organising LTL monitors over distributed systems with a global clock*, Formal Methods in System Design **49** (2016), no. 1-2, 109–158.

Lukas Convent, Sebastian Hungerecker, Martin Leucker, Torben Scheffel, Malte Schmitz, and Daniel Thoma, *Tessla: Temporal stream-based specification language*, CoRR **abs/1808.10717** (2018).

Christian Colombo, Gordon J. Pace, and Gerardo Schneider, *LARVA — Safer Monitoring of Real-Time Java Programs (Tool Paper)*, Seventh IEEE International Conference on Software Engineering and Formal Methods, SEFM 2009, Hanoi, Vietnam, 23-27 November 2009 (Dang Van Hung and Padmanabhan Krishnan, eds.), IEEE Computer Society, 2009, pp. 33–37.

References II

Feng Chen and Grigore Roșu, *Java-MOP: A Monitoring Oriented Programming Environment for Java*, Tools and Algorithms for the Construction and Analysis of Systems, Lecture Notes in Computer Science, Springer, April 2005, pp. 546–550 (en).

Ben D'Angelo, Sriram Sankaranarayanan, César Sánchez, Will Robinson, Bernd Finkbeiner, Henny B. Sipma, Sandeep Mehrotra, and Zohar Manna, *LOLA: runtime monitoring of synchronous systems*, 12th International Symposium on Temporal Representation and Reasoning (TIME 2005), IEEE Computer Society, 2005, pp. 166–174.

Antoine El-Hokayem and Yliès Falcone, *Monitoring decentralized specifications*, in Bultan and Sen [BS17], pp. 125–135.

______, *THEMIS: a tool for decentralized monitoring algorithms*, in Bultan and Sen [BS17], pp. 372–375.

Yliès Falcone, Tom Cornebize, and Jean-Claude Fernandez, *Efficient and generalized decentralized monitoring of regular languages*, Formal Techniques for Distributed Objects, Components, and Systems - 34th IFIP WG 6.1 International Conference, FORTE 2014, Held as Part of the 9th International Federated Conference on Distributed Computing Techniques, DisCoTec 2014, Berlin, Germany, June 3-5, 2014. Proceedings (Erika Ábrahám and Catuscia Palamidessi, eds.), Lecture Notes in Computer Science, vol. 8461, Springer, 2014, pp. 66–83.

Bernd Finkbeiner, Christopher Hahn, Marvin Stenger, and Leander Tentrup, *RVHyper : A Runtime Verification Tool for Temporal Hyperproperties*, Tools and Algorithms for the Construction and Analysis of Systems - 24th International Conference, TACAS 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Proceedings, Part II (Dirk Beyer and Marieke Huisman, eds.), Lecture Notes in Computer Science, vol. 10806, Springer, 2018, pp. 194–200.

Bernd Finkbeiner, Markus N. Rabe, and César Sánchez, *Algorithms for Model Checking HyperLTL and HyperCTL**, Computer Aided Verification - 27th International Conference, CAV 2015, Proceedings, Part I (Daniel Kroening and Corina S. Pasareanu, eds.), Lecture Notes in Computer Science, vol. 9206, Springer, 2015, pp. 30–48.

Paul Gastin and Dietrich Kuske, *Uniform satisfiability problem for local temporal logics over Mazurkiewicz traces*, Inf. Comput. **208** (2010), no. 7, 797–816.

References III

Sylvain Hallé and Raphael Khoury, *Event Stream Processing with BeepBeep 3*, RV-CuBES 2017. An International Workshop on Competitions, Usability, Benchmarks, Evaluation, and Standardisation for Runtime Verification Tools, Kalpa Publications in Computing, vol. 3, EasyChair, 2017, pp. 81–88.

Jeff Huang, Qingzhou Luo, and Grigore Rosu, *Gpredict: Generic predictive concurrency analysis*, 37th IEEE/ACM International Conference on Software Engineering, ICSE 2015, Volume 1, 2015, pp. 847–857.

Jeff Huang, Patrick O'Neil Meredith, and Grigore Rosu, *Maximal sound predictive race detection with control flow abstraction*, Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, ACM, 2014, pp. 337–348.

Klaus Havelund and Grigore Roşu, *An Overview of the Runtime Verification Tool Java PathExplorer*, Formal Methods in System Design **24** (2004), no. 2, 189–215 (en).

Martin Leucker, César Sánchez, Torben Scheffel, Malte Schmitz, and Alexander Schramm, *TeSSLa: runtime verification of non-synchronized real-time streams*, Proceedings of the 33rd Annual ACM Symposium on Applied Computing, SAC 2018, Pau, France, April 09-13, 2018 (Hisham M. Haddad, Roger L. Wainwright, and Richard Chbeir, eds.), ACM, 2018, pp. 1925–1933.

Kamal Lodaya and Pascal Weil, *Rationality in algebras with a series operation*, Inf. Comput. **171** (2001), no. 2, 269–293.

Antoni W. Mazurkiewicz, *Trace theory*, Petri Nets: Central Models and Their Properties, Advances in Petri Nets 1986, Part II, Proceedings of an Advanced Course, Bad Honnef, Germany, 8-19 September 1986 (Wilfried Brauer, Wolfgang Reisig, and Grzegorz Rozenberg, eds.), Lecture Notes in Computer Science, vol. 255, Springer, 1986, pp. 279–324.

Jeremy Manson, William Pugh, and Sarita V. Adve, *The Java Memory Model*, Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '05, ACM, 2005, pp. 378–391.

B. Meenakshi and Ramaswamy Ramanujam, *Reasoning about layered message passing systems*, Computer Languages, Systems & Structures **30** (2004), no. 3-4, 171–206.

Aravind Natarajan, Himanshu Chauhan, Neeraj Mittal, and Vijay K. Garg, *Efficient abstraction algorithms for predicate detection*, Theor. Comput. Sci. **688** (2017), 24–48.