Robustness against Relaxed Memory Models

Memory Models

Roland Meyer

Technische Universität Kaiserslautern
Concurrent Programs with Shared Memory

- Finite number of shared variables \( \{x, y, x_1, \ldots\} \)
- Finite data domain \( \{d, d_0, d_1, \ldots\} \)
- Finite number of finite-control threads \( T_1, \ldots, T_n \) with operations:
  \[ w(x, d), \quad r(x, d) \]

\[
x = y = 0
\]

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>( a : x = 1 )</td>
<td>( p : y = 1 )</td>
</tr>
<tr>
<td>( b : if(y == 0){</td>
<td>( q : if(x == 0){</td>
</tr>
<tr>
<td>( c : \text{crit. sect. 1} )</td>
<td>( r : \text{crit. sect. 2} )</td>
</tr>
<tr>
<td>( d : }</td>
<td>( s : }</td>
</tr>
</tbody>
</table>

Dekker’s mutual exclusion protocol.
**Sequential Consistency (SC) Semantics [Lamport 1979]**

- Threads directly write to and read from memory
- Classical **interleaving semantics**
  - Computations of different threads are **shuffled**
  - Program order is **preserved** for each thread

\[ x = y = 0 \]

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
<th>Mem</th>
</tr>
</thead>
<tbody>
<tr>
<td>( a : x = 1 )</td>
<td>( p : y = 1 )</td>
<td>( x )</td>
</tr>
<tr>
<td>( b : if(y == 0) { )</td>
<td>( q : if(x == 0) { )</td>
<td>( pc = a )</td>
</tr>
<tr>
<td>( c : crit. sect. 1 )</td>
<td>( r : crit. sect. 2 )</td>
<td>( y )</td>
</tr>
<tr>
<td>( d : )</td>
<td>( s : )</td>
<td>( pc = p )</td>
</tr>
<tr>
<td>( )</td>
<td>( )</td>
<td>( 0 )</td>
</tr>
</tbody>
</table>
Sequential Consistency (SC) Semantics [Lamport 1979]

- Threads directly write to and read from memory
- Classical **interleaving semantics**
  - Computations of different threads are **shuffled**
  - Program order is **preserved** for each thread

\[ x = y = 0 \]

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
<th>Mem</th>
</tr>
</thead>
<tbody>
<tr>
<td>( a ): ( x = 1 )</td>
<td>( b ): ( if(y == 0){ )</td>
<td></td>
</tr>
<tr>
<td>( b ): ( if(y == 0){ )</td>
<td>( c ): ( crit. sect. 1 )</td>
<td></td>
</tr>
<tr>
<td>( d ): }</td>
<td>( r ): ( crit. sect. 2 )</td>
<td></td>
</tr>
<tr>
<td>( p ): ( y = 1 )</td>
<td>( s ): }</td>
<td></td>
</tr>
</tbody>
</table>

\( w(x, 1) \)
Sequential Consistency (SC) Semantics [Lamport 1979]

- Threads directly write to and read from memory
- Classical **interleaving semantics**
  - Computations of different threads are shuffled
  - Program order is preserved for each thread

\[
x = y = 0
\]

\[
\begin{array}{c|c|c}
\text{Thread 1} & \text{Thread 2} & \text{Mem} \\
\hline
a : x = 1 & p : y = 1 & x \\
b : \text{if}(y == 0)\{} & q : \text{if}(x == 0)\{} & \color{gray}{1} \\
c : \text{crit. sect. 1} & r : \text{crit. sect. 2} & \color{gray}{y} \\
d : \} & s : \} & \color{gray}{0} \\
\end{array}
\]

\[w(x, 1) \cdot r(y, 0)\]
Sequential Consistency (SC) Semantics [Lamport 1979]

- Threads directly write to and read from memory
- Classical **interleaving semantics**
  - Computations of different threads are shuffled
  - Program order is preserved for each thread

\[
x = y = 0
\]

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
<th>Mem</th>
</tr>
</thead>
<tbody>
<tr>
<td>( a : x = 1 )</td>
<td>( p : y = 1 )</td>
<td>( x )</td>
</tr>
<tr>
<td>( b : \text{if}(y == 0){ )</td>
<td>( q : \text{if}(x == 0){ )</td>
<td>( pc = c )</td>
</tr>
<tr>
<td>( c : \text{crit. sect. 1} )</td>
<td>( r : \text{crit. sect. 2} )</td>
<td>( y )</td>
</tr>
<tr>
<td>( d : } )</td>
<td>( s : } )</td>
<td>( pc = q )</td>
</tr>
</tbody>
</table>

\[w(x, 1) \cdot r(y, 0) \cdot w(y, 1)\]
Sequential Consistency (SC) Semantics [Lamport 1979]

- Threads directly write to and read from memory
- Classical **interleaving semantics**
  - Computations of different threads are **shuffled**
  - Program order is **preserved** for each thread

\[ x = y = 0 \]

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
<th>Mem</th>
</tr>
</thead>
<tbody>
<tr>
<td>( a : x = 1 )</td>
<td>( p : y = 1 )</td>
<td></td>
</tr>
<tr>
<td>( b : if(y == 0) )</td>
<td>( q : if(x == 0) )</td>
<td></td>
</tr>
<tr>
<td>( c : crit. sect. 1 )</td>
<td>( r : crit. sect. 2 )</td>
<td></td>
</tr>
<tr>
<td>( d : } )</td>
<td>( s : } )</td>
<td></td>
</tr>
</tbody>
</table>

\[ w(x, 1) \cdot r(y, 0) \cdot w(y, 1) \cdot \xi \]

Mutual exclusion holds!
Total Store Ordering (TSO) Semantics [SPARC 1994, x86]

- Sequential Consistency forbids compiler and hardware optimizations
- Hence is not implemented by any processor
- Processors have various buffers to reduce latency of memory accesses
- Behavior captured by relaxed memory models
- Here: Total Store Ordering (TSO) memory model
**Total Store Ordering (TSO) Semantics [SPARC 1994, x86]**

- TSO architectures have **write buffers** (FIFO)
- Read takes value from memory if no write to that variable is buffered
- Otherwise read value of last write to that variable in the buffer

\[ x = y = 0 \]

<table>
<thead>
<tr>
<th></th>
<th>Thread 1</th>
<th>Thread 2</th>
<th>Mem</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Thread 1</strong></td>
<td><strong>Thread 2</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(a: x = 1)</td>
<td>(pc = a)</td>
<td></td>
<td>(x)</td>
</tr>
<tr>
<td>(b: if(y == 0))</td>
<td>(p: y = 1)</td>
<td></td>
<td>(0)</td>
</tr>
<tr>
<td>(c: \text{crit. sect. 1})</td>
<td>(q: if(x == 0))</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(d: )</td>
<td>(r: \text{crit. sect. 2})</td>
<td></td>
<td>(y)</td>
</tr>
<tr>
<td></td>
<td>(s: )</td>
<td></td>
<td>(0)</td>
</tr>
</tbody>
</table>

\[ \text{Mem} \]

\[ x \]

\[ y \]

\[ 0 \]
Total Store Ordering (TSO) Semantics [SPARC 1994, x86]

- TSO architectures have write buffers (FIFO)
- Read takes value from memory if no write to that variable is buffered
- Otherwise read value of last write to that variable in the buffer

\[ x = y = 0 \]

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
<th>Mem</th>
</tr>
</thead>
<tbody>
<tr>
<td>( a : x = 1 )</td>
<td></td>
<td></td>
</tr>
<tr>
<td>( b : if (y == 0) { )</td>
<td>( p : y = 1 )</td>
<td></td>
</tr>
<tr>
<td>( c : \text{crit. sect. 1} )</td>
<td>( q : if (x == 0) { )</td>
<td></td>
</tr>
<tr>
<td>( d : } )</td>
<td>( r : \text{crit. sect. 2} )</td>
<td></td>
</tr>
<tr>
<td></td>
<td>( s : } )</td>
<td></td>
</tr>
</tbody>
</table>

\( pc = b \)  \( w(x, 1) \)  \( x \)  \( 0 \)
\( pc = p \)  \( y \)  \( 0 \)

\( isu \)
Total Store Ordering (TSO) Semantics [SPARC 1994, x86]

- TSO architectures have **write buffers** (FIFO)
- Read takes value from memory if no write to that variable is buffered
- Otherwise read value of last write to that variable in the buffer

\[ x = y = 0 \]

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>( a : x = 1 )</td>
<td>( p : y = 1 )</td>
</tr>
<tr>
<td>( b : if (y == 0) { )</td>
<td>( q : if (x == 0) { )</td>
</tr>
<tr>
<td>( c : \text{crit. sect. 1} )</td>
<td>( r : \text{crit. sect. 2} )</td>
</tr>
<tr>
<td>( d : } )</td>
<td>( s : } )</td>
</tr>
</tbody>
</table>

\( w(x, 1) \)

\( x \)

\( 0 \)

\( y \)

\( 0 \)

\( isu \cdot r(y, 0) \)
Total Store Ordering (TSO) Semantics [SPARC 1994, x86]

- TSO architectures have **write buffers** (FIFO)
- Read takes value from memory if no write to that variable is buffered
- Otherwise read value of last write to that variable in the buffer

\[ x = y = 0 \]

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>( a : x = 1 )</td>
<td>( p : y = 1 )</td>
</tr>
<tr>
<td>( b : if(y == 0) )</td>
<td>( q : if(x == 0) )</td>
</tr>
<tr>
<td>( c : \text{crit. sect. 1} )</td>
<td>( r : \text{crit. sect. 2} )</td>
</tr>
<tr>
<td>( d : )</td>
<td>( s : )</td>
</tr>
</tbody>
</table>

\[ \text{isu} \cdot r(y, 0) \cdot \text{isu} \]
Total Store Ordering (TSO) Semantics [SPARC 1994, x86]

- TSO architectures have write buffers (FIFO)
- Read takes value from memory if no write to that variable is buffered
- Otherwise read value of last write to that variable in the buffer

\[ x = y = 0 \]

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>( a : x = 1 )</td>
<td>( p : y = 1 )</td>
</tr>
<tr>
<td>( b : \text{if}(y == 0) { )</td>
<td>( q : \text{if}(x == 0) { )</td>
</tr>
<tr>
<td>( c : \text{crit. sect. 1} )</td>
<td>( r : \text{crit. sect. 2} )</td>
</tr>
<tr>
<td>( d : } )</td>
<td>( s : } )</td>
</tr>
</tbody>
</table>

\[ \text{Mem} \]

\[ x \]

\[ w(x, 1) \]

\[ x \]

\[ 0 \]

\[ y \]

\[ 1 \]

\[ \text{isu} \cdot r(y, 0) \cdot \text{isu} \cdot w(y, 1) \]
Total Store Ordering (TSO) Semantics [SPARC 1994, x86]

- TSO architectures have write buffers (FIFO)
- Read takes value from memory if no write to that variable is buffered
- Otherwise read value of last write to that variable in the buffer

\[ x = y = 0 \]

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>( a : x = 1 )</td>
<td>( p : y = 1 )</td>
</tr>
<tr>
<td>( b : if(y == 0) { )</td>
<td>( q : if(x == 0) { )</td>
</tr>
<tr>
<td>( c : \text{crit. sect. 1} )</td>
<td>( r : \text{crit. sect. 2} )</td>
</tr>
<tr>
<td>( d : } )</td>
<td>( s : } )</td>
</tr>
</tbody>
</table>

Thread 1
\[ pc = c \]
\[ w(x, 1) \]
\[ Mem \]
\( x \)
\( 0 \)

Thread 2
\[ pc = r \]
\[ y \]
\( 1 \)

\[ isu \cdot r(y, 0) \cdot isu \cdot w(y, 1) \cdot r(x, 0) \]
Total Store Ordering (TSO) Semantics [SPARC 1994, x86]

- TSO architectures have write buffers (FIFO)
- Read takes value from memory if no write to that variable is buffered
- Otherwise read value of last write to that variable in the buffer

\[
x = y = 0
\]

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>(a : x = 1)</td>
<td>(p : y = 1)</td>
</tr>
<tr>
<td>(b : if(y == 0))</td>
<td>(q : if(x == 0))</td>
</tr>
<tr>
<td>(c : \text{crit. sect. 1})</td>
<td>(r : \text{crit. sect. 2})</td>
</tr>
<tr>
<td>(d : }</td>
<td>(s : }</td>
</tr>
</tbody>
</table>

\[
isu \cdot r(y, 0) \cdot isu \cdot w(y, 1) \cdot r(x, 0) \cdot w(x, 1)
\]
Total Store Ordering (TSO) Semantics [SPARC 1994, x86]

- TSO architectures have **write buffers** (FIFO)
- Read takes value from memory if no write to that variable is buffered
- Otherwise read value of last write to that variable in the buffer

\[
x = y = 0
\]

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>(a: x = 1)</td>
<td>(p: y = 1)</td>
</tr>
<tr>
<td>(b: \text{if}(y == 0)) {</td>
<td>(q: \text{if}(x == 0)) {</td>
</tr>
<tr>
<td>(c: \text{crit. sect. 1})</td>
<td>(r: \text{crit. sect. 2})</td>
</tr>
<tr>
<td>(d: )</td>
<td>(s: )</td>
</tr>
</tbody>
</table>

\[
isu \cdot r(y, 0) \cdot isu \cdot w(y, 1) \cdot r(x, 0) \cdot w(x, 1)
\]

*Mutual exclusion fails!!!*
Total Store Ordering (TSO) Semantics [SPARC 1994, x86]

- TSO architectures have **write buffers** (FIFO)
- Read takes value from memory if no write to that variable is buffered
- Otherwise read value of last write to that variable in the buffer

\[
x = y = 0
\]

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>(a: x = 1)</td>
<td>(p: y = 1)</td>
</tr>
<tr>
<td>(b: if(y == 0))</td>
<td>(q: if(x == 0))</td>
</tr>
<tr>
<td>(c: \text{crit. sect. 1})</td>
<td>(r: \text{crit. sect. 2})</td>
</tr>
<tr>
<td>(d: )</td>
<td>(s: )</td>
</tr>
</tbody>
</table>

\[
isu \cdot r(y, 0) \cdot isu \cdot w(y, 1) \cdot r(x, 0) \cdot w(x, 1)
\]
Verification Required?!

Relaxed executions may lead to bad behavior.
Verification Required?!

Relaxed executions may lead to bad behavior

If this is the real world, why does anything work?
Verification Required?!

Relaxed executions may lead to bad behavior

If this is the real world, why does anything work?

Theorem [Adve, Hill 1993]: If a program is data-race-free, then SC and TSO semantics coincide.
Verification Required?!

Relaxed executions may lead to bad behavior

If this is the real world, why does anything work?

Theorem [Adve, Hill 1993]: If a program is data-race-free, then SC and TSO semantics coincide.

So, go and write data-race-free programs!
Verification Required?! 

Relaxed executions may lead to bad behavior

If this is the real world, why does anything work?

**Theorem [Adve, Hill 1993]**: If a program is data-race-free, then SC and TSO semantics coincide.

So, go and write data-race-free programs!

Works in 90% of the cases
Verification Required?!

Relaxed executions may lead to bad behavior

If this is the real world, why does anything work?

**Theorem [Adve, Hill 1993]:** If a program is data-race-free, then SC and TSO semantics coincide.

So, go and write data-race-free programs!

Works in 90% of the cases

**Performance-critical code does have data races**
Verification Required?!

Relaxed executions may lead to bad behavior

**If this is the real world, why does anything work?**

**Theorem [Adve, Hill 1993]:** If a program is data-race-free, then SC and TSO semantics coincide.

So, go and write data-race-free programs!

Works in 90% of the cases

**Performance-critical code does have data races**

Concurrency libraries Operating systems HPC@Fraunhofer ITWM
Verification Required?!

Relaxed executions may lead to bad behavior

If this is the real world, why does anything work?

Theorem [Adve, Hill 1993]: If a program is data-race-free, then SC and TSO semantics coincide.

So, go and write data-race-free programs!

Works in 90% of the cases

Performance-critical code does have data races

Concurrency libraries Operating systems HPC@Fraunhofer ITWM

This is where our verification techniques apply
Robustness

Idea: SC semantics is specification
Robustness

Idea: SC semantics is specification

- Relaxed behavior may contain bugs because programmers only had SC in mind
Robustness

Idea: SC semantics is specification

- **Relaxed behavior** may contain bugs because programmers only had SC in mind
- Every relaxed behavior has an SC equivalent (up to traces)
Robustness

Idea: SC semantics is specification

- Relaxed behavior may contain bugs because programmers only had SC in mind
- Every relaxed behavior has an SC equivalent (up to traces)
- Every relaxed behavior that deviates from SC is a programming error
Robustness

Idea: SC semantics is specification

- **Relaxed behavior** may contain bugs because programmers only had SC in mind
- Every **relaxed behavior** has an SC equivalent (up to traces)
- Every **relaxed behavior** that deviates from SC is a **programming error**

Robustness Problem against relaxed memory model **RMM**
Robustness

Idea: SC semantics is specification

- Relaxed behavior may contain bugs because programmers only had SC in mind
- Every relaxed behavior has an SC equivalent (up to traces)
- Every relaxed behavior that deviates from SC is a programming error

Robustness Problem against relaxed memory model \( RMM \)

\textbf{Input}: Program \( P \).
Robustness

Idea: SC semantics is specification

- Relaxed behavior may contain bugs because programmers only had SC in mind
- Every relaxed behavior has an SC equivalent (up to traces)
- Every relaxed behavior that deviates from SC is a programming error

Robustness Problem against relaxed memory model $RMM$

Input: Program $P$.

Problem: Does $\text{Traces}_{RMM}(P) \subseteq \text{Traces}_{SC}(P)$ hold?
Robustness

Idea: SC semantics is specification

- Relaxed behavior may contain bugs because programmers only had SC in mind
- Every relaxed behavior has an SC equivalent (up to traces)
- Every relaxed behavior that deviates from SC is a programming error

Robustness Problem against relaxed memory model $RMM$

Input: Program $P$.

Problem: Does $\text{Traces}_{RMM}(P) \subseteq \text{Traces}_{SC}(P)$ hold?

Decidability / Complexity?
Outline

1. Shared Memory Concurrency
   - Sequential Consistency Semantics
   - Total Store Ordering Semantics

2. Robustness: General Solution
   - Combinatorics
   - Algorithmics

3. Robustness: Efficient Solution
   - Combinatorics
Robustness: General Solution

[Calin, Derevenetc, Majumdar, M., FSTTCS’13]

[Derevenetc, M., ICALP’14]
Robustness: General Solution

[Calin, Derevenetc, Majumdar, M., FSTTCS’13]

[Derevenetc, M., ICALP’14]

Decision procedure for robustness that
Robustness: General Solution

[Calin, Derevenetc, Majumdar, M., FSTTCS’13]

[Derevenetc, M., ICALP’14]

Decision procedure for robustness that

- applies to most memory models (checked TSO, PSO, PGAS, Power)
Robustness: General Solution

[Calin, Derevenetc, Majumdar, M., FSTTCS’13]

[Derevenetc, M., ICALP’14]

Decision procedure for robustness that

- applies to most memory models (checked TSO, PSO, PGAS, Power)
- gives precise complexity
Robustness: General Solution

[Calin, Derevenetc, Majumdar, M., FSTTCS’13]

[Derevenetc, M., ICALP’14]

**Decision procedure** for robustness that

- applies to most memory models (checked TSO, PSO, PGAS, Power)
- gives precise complexity
- ... but relies on a new automaton model and lots of guessing
Robustness: General Solution

Robust Computations

Minimal Violations $= \emptyset$?

RMM-computations

Combinatorics: Violations can be assumed to be in normal form

Algorithmics: Check whether normal form violations exist

Together: Reduce robustness to an emptiness check

$\mathcal{L}_{nf} \cap \mathcal{R}_{cyc} = \emptyset$. 

Roland Meyer (TU KL)
Robustness against Relaxed Memory Models
MM February 2015 10 / 27
Robustness: General Solution

Robust Computations

Minimal Violations $= \emptyset$?

RMM-computations

Combinatorics: Violations can be assumed to be in normal form
Robustness: General Solution

Combinatorics: Violations can be assumed to be in normal form

Algorithmics: Check whether normal form violations exist
Robustness: General Solution

Combinatorics: Violations can be assumed to be in normal form

Algorithmics: Check whether normal form violations exist

Together: Reduce robustness to an emptiness check

\[ L_{nf} \cap R_{cyc} = \emptyset. \]
Robustness: General Solution

Reduce robustness to an emptiness check

\[ \mathcal{L}_{nf} \cap \mathcal{R}_{cyc} \neq \emptyset. \]
Robustness: General Solution

Reduce robustness to an emptiness check

\[ \mathcal{L}_{nf} \cap \mathcal{R}_{cyc} \neq \emptyset. \]

Combinatorics:
Robustness: General Solution

Reduce robustness to an emptiness check

\[ \mathcal{L}_{nf} \cap \mathcal{R}_{cyc} \neq \emptyset. \]

Combinatorics:

- Violations to SC (if any) have a representative in normal form.
Robustness: General Solution

Reduce robustness to an emptiness check

\[ \mathcal{L}_{nf} \cap \mathcal{R}_{cyc} \neq \emptyset. \]

Combinatorics:

- Violations to SC (if any) have a representative in normal form.

Algorithmics:
Robustness: General Solution

Reduce robustness to an emptiness check

\[ \mathcal{L}_{nf} \cap \mathcal{R}_{cyc} \neq \emptyset. \]

Combinatorics:
- Violations to SC (if any) have a representative in normal form.

Algorithmics:
- Language \( \mathcal{L}_{nf} \) consists of all normal-form computations.
Robustness: General Solution

Reduce robustness to an emptiness check

\[ \mathcal{L}_{nf} \cap \mathcal{R}_{cyc} \neq \emptyset. \]

Combinatorics:

- Violations to SC (if any) have a representative in normal form.

Algorithmics:

- Language \( \mathcal{L}_{nf} \) consists of all normal-form computations.
- \( \cap \mathcal{R}_{cyc} \) filters only violating computations.
Robustness: General Solution

Reduce robustness to an emptiness check

\[ \mathcal{L}_{nf} \cap \mathcal{R}_{cyc} \not= \emptyset. \]

Combinatorics:

- Violations to SC (if any) have a representative in normal form.

Algorithmics:

- Language \( \mathcal{L}_{nf} \) consists of all normal-form computations.
- \( \mathcal{L}_{nf} \cap \mathcal{R}_{cyc} \) filters only violating computations.
- Decide \( \mathcal{L}_{nf} \cap \mathcal{R}_{cyc} \not= \emptyset. \)
Robustness: General Solution

Reduce robustness to an emptiness check

\[ \mathcal{L}_{nf} \cap \mathcal{R}_{cyc} \neq \emptyset. \]

Combinatorics:

- Violations to SC (if any) have a representative in normal form.

Algorithmics:

- Language \( \mathcal{L}_{nf} \) consists of all normal-form computations.
- \( \cap \mathcal{R}_{cyc} \) filters only violating computations.
- Decide \( \mathcal{L}_{nf} \cap \mathcal{R}_{cyc} \neq \emptyset. \)
Combinatorics: Normal Form Violations

Lemma (Shasha and Snir, 1988)
A computation violates SC iff it has a cyclic happens-before relation.

\[
\tau = i_s u \cdot r(y, 0) \cdot i_s u \cdot w(y, 1) \cdot r(x, 0) \cdot w(x, 1)
\]

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>( i_{init_x} )</td>
<td>( i_{init_x} )</td>
</tr>
<tr>
<td>( a: w(x, 1) )</td>
<td>( d: r(x, 0) )</td>
</tr>
<tr>
<td>( b: r(y, 0) )</td>
<td>( c: w(y, 1) )</td>
</tr>
</tbody>
</table>
Lemma (Shasha and Snir, 1988)
A computation violates SC iff it has a cyclic happens-before relation.

Happens-before relation of computation

\[ \tau = isu \cdot r(y, 0) \cdot isu \cdot w(y, 1) \cdot r(x, 0) \cdot w(x, 1) : \]

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>init_x</td>
<td>a: w(x, 1)</td>
</tr>
<tr>
<td>init_y</td>
<td>b: r(y, 0)</td>
</tr>
</tbody>
</table>
Lemma (Shasha and Snir, 1988)
A computation violates SC iff it has a cyclic happens-before relation.

Happens-before relation of computation

\[ \tau = \text{isu} \cdot r(y, 0) \cdot \text{isu} \cdot w(y, 1) \cdot r(x, 0) \cdot w(x, 1) : \]

Program order

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{init}_x )</td>
<td>( \text{init}_y )</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Lemma (Shasha and Snir, 1988)
A computation violates SC iff it has a cyclic happens-before relation.

Happens-before relation of computation

\[ \tau = isu \cdot r(y, 0) \cdot isu \cdot w(y, 1) \cdot r(x, 0) \cdot w(x, 1) : \]

Program order, store order

Thread 1
- \text{init}_x 
- \text{a: } w(x, 1)
- \text{po}
- \text{b: } r(y, 0)

Thread 2
- \text{init}_y 
- \text{c: } w(y, 1)
- \text{d: } r(x, 0)
- \text{po}
Lemma (Shasha and Snir, 1988)
A computation violates SC iff it has a cyclic happens-before relation.

Happens-before relation of computation

\[
\tau = \text{isu} \cdot r(y, 0) \cdot \text{isu} \cdot w(y, 1) \cdot r(x, 0) \cdot w(x, 1) : 
\]

Program order, store order, source relation
Combinatorics: Normal Form Violations

**Lemma (Shasha and Snir, 1988)**
A computation violates SC iff it has a **cyclic** happens-before relation.

**Happens-before relation of computation**

$$\tau = isu \cdot r(y, 0) \cdot isu \cdot w(y, 1) \cdot r(x, 0) \cdot w(x, 1) :$$

Program order, store order, source relation, conflict relation
Combinatorics: Normal Form Violations

Normal Form:

\[ \tau = \tau_1 \cdot \tau_2 \]

Delays in \( \tau_2 \) respect ordering in \( \tau_1 \)

In normal form...

\( \tau_{1} \).

\( w(x, 1) \).

\( w(y, 1) \).

Not in normal form...

\( \tau_{1} \).

\( w(y, 1) \).

\( w(x, 1) \).
Combinatorics: Normal Form Violations

Normal Form:
- Computation has two parts $\tau = \tau_1 \cdot \tau_2$
Combinatorics: Normal Form Violations

Normal Form:

- Computation has two parts $\tau = \tau_1 \cdot \tau_2$
- Delays in $\tau_2$ respect ordering in $\tau_1$
Combinatorics: Normal Form Violations

Normal Form:
- Computation has two parts $\tau = \tau_1 \cdot \tau_2$
- Delays in $\tau_2$ respect ordering in $\tau_1$

In normal form

\[ \ldots isu \ldots isu \ldots \underbrace{\ldots w(x,1) \ldots w(y,1)}_{\tau_2} \ldots \underbrace{\ldots \tau_1 \ldots}_{\tau_1} \ldots \]
Combinatorics: Normal Form Violations

Normal Form:
- Computation has two parts $\tau = \tau_1 \cdot \tau_2$
- Delays in $\tau_2$ respect ordering in $\tau_1$

In normal form

\[
\begin{align*}
\cdots & \text{isu} \cdots \text{isu} \cdots \\
& \tau_1 & \quad & \quad \tau_2 \\
\quad & \text{w}(x, 1) \cdots \text{w}(y, 1) \cdots
\end{align*}
\]

Not in normal form

\[
\begin{align*}
\cdots & \text{isu} \cdots \text{isu} \cdots \\
& \tau_1 & \quad & \quad \tau_2 \\
\quad & \text{w}(y, 1) \cdots \text{w}(x, 1) \cdots
\end{align*}
\]
Combinatorics: Normal Form Violations

**Theorem (Normal form):**
If a program is not robust, it has a violation in normal form.

Proof:
Take a shortest computation \( \tau \) with cyclic happens-before relation. There is (may be non-trivial, depending on RMM) an event that can be cancelled:
\[ \tau = \tau_1 \cdot a \cdot \tau_2. \]
Computation \( \tau_1 \cdot \tau_2 \) is shorter, hence not violating.
There is an SC computation \( \sigma \) with same happens-before relation.
Now \( (\sigma \downarrow \tau_1) \cdot a \cdot (\sigma \downarrow \tau_2) \) is in normal form and violating.
Combinatorics: Normal Form Violations

Theorem (Normal form):
If a program is not robust, it has a violation in normal form.

Proof:
- Take a shortest computation $\tau$ with cyclic happens-before relation.
Combinatorics: Normal Form Violations

Theorem (Normal form):
If a program is not robust, it has a violation in normal form.

Proof:
• Take a shortest computation \( \tau \) with cyclic happens-before relation.
• There is (may be non-trivial, depending on RMM) an event that can be cancelled:

\[
\tau = \tau_1 \cdot a \cdot \tau_2 .
\]
**Theorem (Normal form):**
If a program is not robust, it has a violation in normal form.

**Proof:**
- Take a shortest computation $\tau$ with cyclic happens-before relation.
- There is (may be non-trivial, depending on RMM) an event that can be cancelled:
  \[ \tau = \tau_1 \cdot a \cdot \tau_2 . \]
- Computation $\tau_1 \cdot \tau_2$ is shorter.
Combinatorics: Normal Form Violations

**Theorem (Normal form):**
If a program is not robust, it has a violation in normal form.

**Proof:**
- Take a shortest computation $\tau$ with cyclic happens-before relation.
- There is (may be non-trivial, depending on RMM) an event that can be cancelled:
  \[ \tau = \tau_1 \cdot a \cdot \tau_2 \cdot \]
- Computation $\tau_1 \cdot \tau_2$ is shorter, hence *not violating*.
Combinatorics: Normal Form Violations

**Theorem (Normal form):**
If a program is not robust, it has a violation in normal form.

**Proof:**
- Take a shortest computation $\tau$ with cyclic happens-before relation.
- There is (may be non-trivial, depending on RMM) an event that can be cancelled:

$$\tau = \tau_1 \cdot a \cdot \tau_2 .$$

- Computation $\tau_1 \cdot \tau_2$ is shorter, hence not violating.
- There is an SC computation $\sigma$ with same happens-before relation.
Combinatorics: Normal Form Violations

Theorem (Normal form):
If a program is not robust, it has a violation in normal form.

Proof:
- Take a shortest computation \( \tau \) with cyclic happens-before relation.
- There is (may be non-trivial, depending on RMM) an event that can be cancelled:
  \[ \tau = \tau_1 \cdot a \cdot \tau_2. \]
  
  Computation \( \tau_1 \cdot \tau_2 \) is shorter, hence not violating.
- There is an SC computation \( \sigma \) with same happens-before relation.
- Now
  \[ (\sigma \downarrow \tau_1) \cdot a \cdot (\sigma \downarrow \tau_2) \]
Combinatorics: Normal Form Violations

Theorem (Normal form):
If a program is not robust, it has a violation in normal form.

Proof:
- Take a shortest computation $\tau$ with cyclic happens-before relation.
- There is (may be non-trivial, depending on RMM) an event that can be cancelled:

$$\tau = \tau_1 \cdot a \cdot \tau_2 .$$

- Computation $\tau_1 \cdot \tau_2$ is shorter, hence not violating.
- There is an SC computation $\sigma$ with same happens-before relation.
- Now

$$\left(\sigma \downarrow \tau_1\right) \cdot a \cdot \left(\sigma \downarrow \tau_2\right)$$

is in normal form
Combinatorics: Normal Form Violations

**Theorem (Normal form):**
If a program is not robust, it has a violation in normal form.

**Proof:**
- Take a shortest computation $\tau$ with cyclic happens-before relation.
- There is (may be non-trivial, depending on RMM) an event that can be cancelled:

$$\tau = \tau_1 \cdot a \cdot \tau_2 .$$

- Computation $\tau_1 \cdot \tau_2$ is shorter, hence not violating.
- There is an SC computation $\sigma$ with same happens-before relation.
- Now

$$\left( \sigma \downarrow \tau_1 \right) \cdot a \cdot \left( \sigma \downarrow \tau_2 \right)$$

is in normal form and violating.
Robustness: General Solution

Reduce robustness to an emptiness check

\[ \mathcal{L}_{nf} \cap \mathcal{R}_{cyc} \neq \emptyset. \]

Combinatorics:

- Violations to SC (if any) have a representative in normal form.

Algorithmics:

- Language \( \mathcal{L}_{nf} \) consists of all normal-form computations.
- \( \cap \mathcal{R}_{cyc} \) filters only violating computations.
- Decide \( \mathcal{L}_{nf} \cap \mathcal{R}_{cyc} \neq \emptyset. \)
Challenge
Describe language $\mathcal{L}_{nf}$ of all normal-form computations
Challenge
Describe language $\mathcal{L}_{nf}$ of all normal-form computations

Need a language class that
Algorithmics: Generating Normal-Form Computations

**Challenge**
Describe language $\mathcal{L}_{nf}$ of all normal-form computations

Need a language class that
- includes $\mathcal{L}_{nf}$,
Algorithmics: Generating Normal-Form Computations

**Challenge**
Describe language $\mathcal{L}_{nf}$ of all normal-form computations

Need a language class that
- includes $\mathcal{L}_{nf}$,
- is closed under regular intersection $(\mathcal{L}_{nf} \cap \mathcal{R}_{cyc})$,
Algorithmics: Generating Normal-Form Computations

**Challenge**
Describe language $\mathcal{L}_{nf}$ of all normal-form computations

Need a language class that
- includes $\mathcal{L}_{nf}$,
- is closed under regular intersection ($\mathcal{L}_{nf} \cap \mathcal{R}_{cyc}$),
- has **decidable emptiness** problem ($\mathcal{L}_{nf} \cap \mathcal{R}_{cyc} \neq \emptyset$).
Algorithmics: Generating Normal-Form Computations

**Challenge**
Describe language $L_{nf}$ of all normal-form computations

Need a language class that
- includes $L_{nf}$,
- is closed under regular intersection ($L_{nf} \cap R_{cyc}$),
- has decidable emptiness problem ($L_{nf} \cap R_{cyc} \neq \emptyset$).

**Properties of $L_{nf}$**
Algorithmics: Generating Normal-Form Computations

**Challenge**
Describe language $\mathcal{L}_{nf}$ of all normal-form computations

Need a language class that
- includes $\mathcal{L}_{nf}$,
- is closed under regular intersection ($\mathcal{L}_{nf} \cap \mathcal{R}_{cyc}$),
- has decidable emptiness problem ($\mathcal{L}_{nf} \cap \mathcal{R}_{cyc} \neq \emptyset$).

**Properties of $\mathcal{L}_{nf}$**
- Number of concurrently executed instructions is unbounded
Algorithmics: Generating Normal-Form Computations

**Challenge**
Describe language $\mathcal{L}_{nf}$ of all normal-form computations

Need a language class that

- includes $\mathcal{L}_{nf}$,
- is closed under regular intersection ($\mathcal{L}_{nf} \cap \mathcal{R}_{cyc}$),
- has decidable emptiness problem ($\mathcal{L}_{nf} \cap \mathcal{R}_{cyc} \neq \emptyset$).

**Properties of $\mathcal{L}_{nf}$**

- Number of concurrently executed instructions is unbounded
- May include computations like $isu^n \cdot w(x,1)^n$
Algorithmics: Generating Normal-Form Computations

**Challenge**

Describe language $\mathcal{L}_{nf}$ of all normal-form computations

Need a language class that

- includes $\mathcal{L}_{nf}$,
- is closed under regular intersection ($\mathcal{L}_{nf} \cap \mathcal{R}_{cyc}$),
- has decidable emptiness problem ($\mathcal{L}_{nf} \cap \mathcal{R}_{cyc} ?= \emptyset$).

**Properties of $\mathcal{L}_{nf}$**

- Number of concurrently executed instructions is unbounded
- May include computations like $isu^n \cdot w(x,1)^n$
  
  $\Rightarrow$ not regular
Solution
Define $L_{nf}$ as language of a multiheaded automaton
Algorithmics: Generating Normal-Form Computations

Solution
Define $\mathcal{L}_{nf}$ as language of a multiheaded automaton

Multiheaded automata

---

\[
\ldots isu \ldots isu \ldots \quad \ldots w(x, 1) \ldots w(y, 1) \ldots
\]
Algorithmics: Generating Normal-Form Computations

**Solution**

Define $\mathcal{L}_{nf}$ as language of a **multiheaded automaton**

Multiheaded automata

- Extension of NFA

---

$\ldots \text{isu} \ldots \text{isu} \ldots$ 

$\tau_1$

$\ldots \text{w}(x, 1) \ldots \text{w}(y, 1) \ldots$

$\tau_2$
Solution
Define \( \mathcal{L}_{nf} \) as language of a multiheaded automaton

Multiheaded automata
- Extension of NFA
- Generates parts \( \tau_1 \) and \( \tau_2 \) of a computation \( \tau_1 \cdot \tau_2 \) simultaneously
Solution
Define \( \mathcal{L}_{nf} \) as language of a multiheaded automaton

Multiheaded automata
- Extension of NFA
- Generates parts \( \tau_1 \) and \( \tau_2 \) of a computation \( \tau_1 \cdot \tau_2 \) simultaneously
- Transitions \( q^{1,a} \rightarrow q' \) and \( q^{2,b} \rightarrow q' \) labeled by head \( i = 1, 2 \)
Algorithmics: Generating Normal-Form Computations

Solution
Define $\mathcal{L}_{nf}$ as language of a multiheaded automaton

Multiheaded automata
- Extension of NFA
- Generates parts $\tau_1$ and $\tau_2$ of a computation $\tau_1 \cdot \tau_2$ simultaneously
- Transitions $q \xrightarrow{1,a} q'$ and $q \xrightarrow{2,b} q'$ labeled by head $i = 1, 2$

Example:

$$
\tau_1 \quad \cdots \text{isu} \cdots \text{isu} \cdots \\
\quad \quad \quad \tau_1 \\
\tau_2 \quad \quad \cdots \text{w}(x,1) \cdots \text{w}(y,1) \cdots \\
\quad \quad \quad \tau_2
$$
Solution

Define $\mathcal{L}_{nf}$ as language of a multiheaded automaton

Multiheaded automata

- Extension of NFA
- Generates parts $\tau_1$ and $\tau_2$ of a computation $\tau_1 \cdot \tau_2$ simultaneously
- Transitions $q \xrightarrow{1,a} q'$ and $q \xrightarrow{2,b} q'$ labeled by head $i = 1, 2$

Example:
Solution
Define \( \mathcal{L}_{nf} \) as language of a multiheaded automaton

Multiheaded automata

- Extension of NFA
- Generates parts \( \tau_1 \) and \( \tau_2 \) of a computation \( \tau_1 \cdot \tau_2 \) simultaneously
- Transitions \( q \xrightarrow{1,a} q' \) and \( q \xrightarrow{2,b} q' \) labeled by head \( i = 1, 2 \)

Example:
Solution
Define $\mathcal{L}_{nf}$ as language of a multiheaded automaton

Multiheaded automata
- Extension of NFA
- Generates parts $\tau_1$ and $\tau_2$ of a computation $\tau_1 \cdot \tau_2$ simultaneously
- Transitions $q \xrightarrow{1,a} q'$ and $q \xrightarrow{2,b} q'$ labeled by head $i = 1, 2$

Example:
Solution
Define $\mathcal{L}_{nf}$ as language of a multiheaded automaton

Multiheaded automata
- Extension of NFA
- Generates parts $\tau_1$ and $\tau_2$ of a computation $\tau_1 \cdot \tau_2$ simultaneously
- Transitions $q \xrightarrow{1,a} q'$ and $q \xrightarrow{2,b} q'$ labeled by head $i = 1, 2$

Example:

Transitions: $q_1 \xrightarrow{1,isu} q_2 \xrightarrow{2,w(x,1)} q_3$
Algorithmics: Generating Normal-Form Computations

**Solution**

Define $\mathcal{L}_{nf}$ as language of a multiheaded automaton

**Multiheaded automata**

- Extension of NFA
- Generates parts $\tau_1$ and $\tau_2$ of a computation $\tau_1 \cdot \tau_2$ simultaneously
- Transitions $q \xrightarrow{1,a} q'$ and $q \xrightarrow{2,b} q'$ labeled by head $i = 1, 2$

Example:

\[
\begin{array}{c}
\ldots isu \uparrow \ldots isu \ldots \\
\tau_1 \\
\ldots \uparrow w(x, 1) \ldots w(y, 1) \ldots \\
\tau_2
\end{array}
\]

Transitions: $q_1 \xrightarrow{1,isu} q_2 \xrightarrow{2,w(x,1)} q_3$
**Solution**

Define $L_{nf}$ as language of a multiheaded automaton.

**Multiheaded automata**

- Extension of NFA
- Generates parts $\tau_1$ and $\tau_2$ of a computation $\tau_1 \cdot \tau_2$ simultaneously
- Transitions $q_{1,a} \rightarrow q'$ and $q_{2,b} \rightarrow q'$ labeled by head $i = 1, 2$

**Example:**

\[
\begin{align*}
&\cdots isu \uparrow \cdots isu \cdots \\
&\hline
&\tau_1 \\
&\cdots w(x, 1) \uparrow \cdots w(y, 1) \cdots \\
&\hline
&\tau_2 \\
\end{align*}
\]

Transitions: $q_1 \xrightarrow{1,isu} q_2 \xrightarrow{2,w(x,1)} q_3$
Robustness: General Solution

Reduce robustness to an emptiness check

\[ \mathcal{L}_{nf} \cap \mathcal{R}_{cyc} \neq \emptyset. \]

Combinatorics:

- Violations to SC (if any) have a representative in normal form.

Algorithmics:

- Language \( \mathcal{L}_{nf} \) consists of all normal-form computations.
- \( \cap \mathcal{R}_{cyc} \) filters only violating computations.
- Decide \( \mathcal{L}_{nf} \cap \mathcal{R}_{cyc} \neq \emptyset. \)
Algorithmics: Checking Cyclicity

Happens-before relation from the example:

Thread 1

\[ \text{init}_x \rightarrow \text{a: } w(x, 1) \]
\[ \text{st} \rightarrow \text{b: } r(y, 0) \]
\[ \text{po} \rightarrow \text{c: } w(y, 1) \]

Thread 2

\[ \text{d: } r(x, 0) \]
\[ \text{cf} \rightarrow \text{src} \]
\[ \text{po} \rightarrow \text{cf} \]

\[ \text{st} \rightarrow \text{src} \]
Algorithmics: Checking Cyclicity

Happens-before relation from the example:

Checking cyclicity
Algorithmics: Checking Cyclicity

Happens-before relation from the example:

Checking cyclicity

- Finitely many types of cycles
Algorithmics: Checking Cyclicity

Happens-before relation from the example:

```
init_x
init_y
```

```
<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>init_x</td>
<td>init_y</td>
</tr>
<tr>
<td>src</td>
<td>src</td>
</tr>
<tr>
<td>st</td>
<td>st</td>
</tr>
<tr>
<td>po</td>
<td>cf</td>
</tr>
<tr>
<td>b: r(y, 0)</td>
<td>d: r(x, 0)</td>
</tr>
<tr>
<td>c: w(y, 1)</td>
<td>cf</td>
</tr>
</tbody>
</table>
```

Checking cyclicity

- Finitely many types of cycles
- Guess per thread two instructions in program order
Algorithmics: Checking Cyclicity

Happens-before relation from the example:

```
init\_x  \rightarrow \text{Thread 1} \quad \text{Thread 2}  \rightarrow \text{po} \quad \text{cf}
\quad \text{src}  \rightarrow \text{st}  \rightarrow \text{st}
```

```
\text{Thread 1}: \quad \begin{align*}
    a & : w(x, 1) \\
    b & : r(y, 0)
\end{align*}
```

```
\text{Thread 2}: \quad \begin{align*}
    d & : r(x, 0) \\
    c & : w(y, 1)
\end{align*}
```

Checking cyclicity

- Finitely many types of cycles
- Guess per thread two instructions in program order
- **Finite automata check edges** between guessed instructions from different threads
Robustness: General Solution

Reduce robustness to an emptiness check

\[ \mathcal{L}_{nf} \cap \mathcal{R}_{cyc} \equiv \emptyset. \]

Combinatorics:
- Violations to SC (if any) have a representative in normal form.

Algorithmics:
- Language \( \mathcal{L}_{nf} \) consists of all normal-form computations.
- \( \cap \mathcal{R}_{cyc} \) filters only violating computations.
- Decide \( \mathcal{L}_{nf} \cap \mathcal{R}_{cyc} \equiv \emptyset \).
Algorithmics: Emptiness

**Theorem:**
Assuming finite memory, robustness is $\text{PSPACE}$-complete.
Theorem:
Assuming finite memory, robustness is \textit{PSPACE}-complete.

Proof:
Theorem:
Assuming finite memory, robustness is \textsc{PSPACE}-complete.

Proof:
\begin{itemize}
  \item Upper bound: $\mathcal{L}_{nf} \cap \mathcal{R}_{cyc} \neq \emptyset$.
\end{itemize}
Algorithmics: Emptiness

Theorem:
Assuming finite memory, robustness is $\text{PSPACE}$-complete.

Proof:
- Upper bound: $\mathcal{L}_{nf} \cap \mathcal{R}_{cyc} \neq \emptyset$.
- Lower bound: SC state reachability [Kozen 1977].
Robustness: Efficient Solution

[Bouajjani, M., Möhlmann, ICALP’11]
[Bouajjani, Derevenetc, M., ESOP’13]
Robustness: Efficient Solution

[Bouajjani, M., Möhlmann, ICALP’11]

[Bouajjani, Derevenetc, M., ESOP’13]

Decision procedure for robustness that
Robustness: Efficient Solution

[Bojaji, M., Möhlmann, ICALP’11]
[Bojaji, Derevenetc, M., ESOP’13]

Decision procedure for robustness that
- uses standard SC reachability in ordinary parallel programs
Robustness: Efficient Solution

[Bouajjani, M., Möhlmann, ICALP’11]
[Bouajjani, Derevenetc, M., ESOP’13]

Decision procedure for robustness that

- uses standard SC reachability in ordinary parallel programs
  (rather than emptiness in multiheaded automata)
Robustness: Efficient Solution

[Bouajjani, M., Möhlmann, ICALP’11]
[Bouajjani, Derevenetc, M., ESOP’13]

Decision procedure for robustness that

- uses standard SC reachability in ordinary parallel programs (rather than emptiness in multiheaded automata)
- avoids non-determinism for finding cycles (regular intersection)
Robustness: Efficient Solution

[Bouajjani, M., Möhlmann, ICALP’11]
[Bouajjani, Derevenetc, M., ESOP’13]

Decision procedure for robustness that

- uses standard SC reachability in ordinary parallel programs
  (rather than emptiness in multiheaded automata)
- avoids non-determinism for finding cycles (regular intersection)
- ... but is hard to apply (so far only works for TSO)
Robustness: Efficient Solution

- Robust Computations
- Minimal Violations $= \emptyset$?
- RMM Computations

Combinatorics: Violations can be assumed to be local
Algorithmics: Check whether local violations exist
Together: Reduce robustness to SC reachability in an instrumented program
Robustness: Efficient Solution

**Combinatorics:** Violations can be assumed to be local
Robustness: Efficient Solution

Combinatorics: Violations can be assumed to be local — one thread delays
Robustness: Efficient Solution

Combinatorics: Violations can be assumed to be local — one thread delays
Algorithmics: Check whether local violations exist
Robustness: Efficient Solution

**Combinatorics:** Violations can be assumed to be local — one thread delays

**Algorithmics:** Check whether local violations exist

**Together:** Reduce robustness to SC reachability in an instrumented program
Combinatorics: Locality Theorem

**Theorem (Locality):**

If a program is not robust against TSO, then there is a violating computation where *only one thread* delays writes.
Combinatorics: Locality Theorem

**Theorem (Locality):**

If a program is not robust against TSO, then there is a violating computation where *only one thread* delays writes.

**Combinatorial set-up:**

Consider a violating computation, where
Combinatorics: Locality Theorem

**Theorem (Locality):**

If a program is not robust against TSO, then there is a violating computation where *only one thread* delays writes.

**Combinatorial set-up:**

Consider a violating computation, where

\[ \tau_1 \cdot a \cdot \tau_2 \cdot b \cdot \tau_3 \]

with\( \text{thread}(a) = \text{thread}(b) \), there is a happens-before path between \( a \) and \( b \) through \( \tau_2 \).

number of delays is minimal.
Combinatorics: Locality Theorem

Theorem (Locality):
If a program is not robust against TSO, then there is a violating computation where only one thread delays writes.

Combinatorial set-up:
Consider a violating computation, where

number of delays is minimal.

Lemma:
In a minimal violation $\tau_1 \cdot a \cdot \tau_2 \cdot b \cdot \tau_3$ with $\text{thread}(a) = \text{thread}(b)$, there is a happens-before path between $a$ and $b$ through $\tau_2$. 
Combinatorics: Locality Theorem

Proof (Locality Theorem):
Three cases:

\[ \ldots \text{isu} \ldots w(y, 1) \ldots \ldots \text{isu} \ldots w(x, 1) \ldots \]

\[ \ldots \text{isu} \ldots \text{isu} \ldots \ldots w(y, 1) \ldots w(y, 1) \ldots \]

\[ \ldots \text{isu} \ldots \text{isu} \ldots \ldots w(x, 1) \ldots w(y, 1) \ldots \]

Non-trivial
Conclusion

Robustness

- Compares relaxed behaviors against SC
Conclusion

Robustness

- Compares relaxed behaviors against SC
- Problem is PSPACE-complete for many models
Conclusion

Robustness

- Compares relaxed behaviors against SC
- Problem is PSPACE-complete for many models
- Upper bound uses unified approach
Conclusion

Robustness

- Compares relaxed behaviors against SC
- Problem is PSPACE-complete for many models
- Upper bound uses unified approach based on normal forms and multiheaded automata
Conclusion

Robustness

- Compares relaxed behaviors against SC
- Problem is PSPACE-complete for many models
- Upper bound uses unified approach based on normal forms and multiheaded automata
- Efficient implementations rely on locality

Alternative: Reachability

Decidable for TSO (and beyond), but non-primitive recursive [Atig et al. POPL'10, ESOP'12]
Abstraction-based techniques [Kuperstein, Vechev, Yahav, PLDI'11]
Symbolic techniques [Abdulla et al., TACAS'12][Linden, Wolper, SPIN'10'11]

Pros: Robustness does not need a spec.
Reachability checks what is needed.
Robustness is cheaper.
Conclusion

Robustness

- Compares relaxed behaviors against SC
- Problem is PSPACE-complete for many models
- Upper bound uses unified approach based on normal forms and multiheaded automata
- Efficient implementations rely on locality

Alternative: Reachability
Conclusion

Robustness

- Compares relaxed behaviors against SC
- Problem is PSPACE-complete for many models
- Upper bound uses unified approach based on normal forms and multiheaded automata
- Efficient implementations rely on locality

Alternative: Reachability

- Decidable for TSO (and beyond), but non-primitive recursive [Atig et al. POPL’10, ESOP’12]
Conclusion

Robustness

- Compares relaxed behaviors against SC
- Problem is PSPACE-complete for many models
- Upper bound uses unified approach based on normal forms and multiheaded automata
- Efficient implementations rely on locality

Alternative: Reachability

- Decidable for TSO (and beyond), but non-primitive recursive [Atig et al. POPL’10, ESOP’12]
- Abstraction-based techniques [Kuperstein, Vechev, Yahav, PLDI’11]
Conclusion

Robustness

- Compares relaxed behaviors against SC
- Problem is PSPACE-complete for many models
- Upper bound uses unified approach based on normal forms and multiheaded automata
- Efficient implementations rely on locality

Alternative: Reachability

- Decidable for TSO (and beyond), but non-primitive recursive [Atig et al. POPL’10, ESOP’12]
- Abstraction-based techniques [Kuperstein, Vechev, Yahav, PLDI’11]
- Symbolic techniques [Abdulla et al., TACAS’12][Linden, Wolper, SPIN’10’11]
Conclusion

Robustness

- Compares relaxed behaviors against SC
- Problem is PSPACE-complete for many models
- Upper bound uses unified approach based on normal forms and multiheaded automata
- Efficient implementations rely on locality

Alternative: Reachability

- Decidable for TSO (and beyond), but non-primitive recursive [Atig et al. POPL’10, ESOP’12]
- Abstraction-based techniques [Kuperstein, Vechev, Yahav, PLDI’11]
- Symbolic techniques [Abdulla et al., TACAS’12][Linden, Wolper, SPIN’10’11]

Pros: Robustness does not need a spec.
Conclusion

Robustness

- Compares relaxed behaviors against SC
- Problem is PSPACE-complete for many models
- Upper bound uses unified approach based on normal forms and multiheaded automata
- Efficient implementations rely on locality

Alternative: Reachability

- Decidable for TSO (and beyond), but non-primitive recursive [Atig et al. POPL’10, ESOP’12]
- Abstraction-based techniques [Kuperstein, Vechev, Yahav, PLDI’11]
- Symbolic techniques [Abdulla et al., TACAS’12][Linden, Wolper, SPIN’10’11]

Pros: Robustness does not need a spec. Reachability checks what is needed.
Conclusion

Robustness

- Compares relaxed behaviors against SC
- Problem is PSPACE-complete for many models
- Upper bound uses unified approach based on normal forms and multiheaded automata
- Efficient implementations rely on locality

Alternative: Reachability

- Decidable for TSO (and beyond), but non-primitive recursive [Atig et al. POPL’10, ESOP’12]
- Abstraction-based techniques [Kuperstein, Vechev, Yahav, PLDI’11]
- Symbolic techniques [Abdulla et al., TACAS’12][Linden, Wolper, SPIN’10’11]

Pros: Robustness does not need a spec. Reachability checks what is needed. Robustness is cheaper.