Just Verification of Mutual Exclusion Algorithms

Rob van Glabbeek, Bas Luttik, and Myrthe Spronck

28 August 2025

Eindhoven University of Technology - Formal System Analysis

TU/e

Mutual exclusion

Mutual exclusion

The problem

- $N \geq 2$ threads, code divided in *critical* and *non-critical* sections
- Leaving non-critical section is optional

```
Algorithm Mutex
  repeat
      non-critical section

      critical section
```

Mutual exclusion

The problem

- $N \geq 2$ threads, code divided in *critical* and *non-critical* sections
- Leaving non-critical section is optional

Properties

- **Mutual exclusion**
- **Deadlock freedom**
- **Starvation freedom**

```
Algorithm Mutex
  repeat
      non-critical section
      entry protocol
      critical section
      exit protocol
```

Mutual exclusion

The problem

- $N \geq 2$ threads, code divided in *critical* and *non-critical* sections
- Leaving non-critical section is optional

Properties

- **Mutual exclusion**
  No two threads in critical section simultaneously
- **Deadlock freedom**
- **Starvation freedom**

```
Algorithm Mutex
  repeat
      non-critical section
      entry protocol
  $\cancel{T_i \wedge T_j}$ → critical section
      exit protocol
```

Mutual exclusion

The problem

- $N \geq 2$ threads, code divided in *critical* and *non-critical* sections
- Leaving non-critical section is optional

Properties

- **Mutual exclusion**
  No two threads in critical section simultaneously
- **Deadlock freedom**
  T_i in entry protocol $\rightarrow$ eventually T_j enters critical section
- **Starvation freedom**

```
Algorithm Mutex
    repeat
          non-critical section
$T_i \longrightarrow$ entry protocol      $T_j$
          critical section
          exit protocol
```

Mutual exclusion

The problem

- $N \geq 2$ threads, code divided in *critical* and *non-critical* sections
- Leaving non-critical section is optional

Properties

- **Mutual exclusion**
  No two threads in critical section simultaneously
- **Deadlock freedom**
  T_i in entry protocol $\rightarrow$ eventually T_j enters critical section
- **Starvation freedom**
  T_i leaves non-critical section $\rightarrow$ eventually T_i enters critical section

```
Algorithm Mutex
  repeat
$T_i$     non-critical section
          entry protocol        $T_i$
          critical section
          exit protocol
```

Mutual exclusion

The problem

- $N \geq 2$ threads, code divided in *critical* and *non-critical* sections
- Leaving non-critical section is optional

Properties

- **Mutual exclusion**
  No two threads in critical section simultaneously
- **Deadlock freedom**
  T_i in entry protocol $\rightarrow$ eventually T_j enters critical section
- **Starvation freedom**
  T_i leaves non-critical section $\rightarrow$ eventually T_i enters critical section

```
Algorithm Mutex
  repeat
      non-critical section
      entry protocol
      critical section
      exit protocol
```

Verification

Goal: verify many mutual exclusion algorithms

Verification

Goal: verify many mutual exclusion algorithms

Method: model checking with mCRL2

- Model algorithms and environment in process-algebra
- Capture properties in modal μ-calculus

Verification

Goal: verify many mutual exclusion algorithms
Method: model checking with mCRL2

- Model algorithms and environment in process-algebra
- Capture properties in modal μ-calculus

Abstraction leads to unrealistic executions $\rightarrow$ must disregard

Verification

Goal: verify many mutual exclusion algorithms
Method: model checking with mCRL2

- Model algorithms and environment in process-algebra
- Capture properties in modal μ-calculus

Abstraction leads to unrealistic executions $\longrightarrow$ must disregard
Which?

Verification

Goal: verify many mutual exclusion algorithms
Method: model checking with mCRL2

- Model algorithms and environment in process-algebra
- Capture properties in modal μ-calculus

Abstraction leads to unrealistic executions $\rightarrow$ must disregard
Which?
Example: starvation freedom violation of Dekker's algorithm

Example violation

Algorithm Dekker's for T_0

```
1: non-critical section
2: $flag[0] \leftarrow 1$
3: while $flag[1] = 1$ do
4:     if $turn = 1$ then
5:         $flag[0] \leftarrow 0$
6:         await $turn = 0$
7:         $flag[0] \leftarrow 1$
8: critical section
9: $turn \leftarrow 1$
10: $flag[0] \leftarrow 0$
```

$$flag[0] = 0$$
$$flag[1] = 0$$
$$turn = 0$$

Algorithm Dekker's for T_1

```
1: non-critical section
2: $flag[1] \leftarrow 1$
3: while $flag[0] = 1$ do
4:     if $turn = 0$ then
5:         $flag[1] \leftarrow 0$
6:         await $turn = 1$
7:         $flag[1] \leftarrow 1$
8: critical section
9: $turn \leftarrow 0$
10: $flag[1] \leftarrow 0$
```

Example violation

Algorithm Dekker's for T_0

```
1: non-critical section
2: $flag[0] \leftarrow 1$
3: while $flag[1] = 1$ do
4:     if $turn = 1$ then
5:         $flag[0] \leftarrow 0$
6:         await $turn = 0$
7:         $flag[0] \leftarrow 1$
8: critical section
9: $turn \leftarrow 1$
10: $flag[0] \leftarrow 0$
```

$flag[0] = 0$

$flag[1] = 0$

$turn = 1$

Algorithm Dekker's for T_1

```
1: non-critical section
2: $flag[1] \leftarrow 1$
3: while $flag[0] = 1$ do
4:     if $turn = 0$ then
5:         $flag[1] \leftarrow 0$
6:         await $turn = 1$
7:         $flag[1] \leftarrow 1$
8: critical section
9: $turn \leftarrow 0$
10: $flag[1] \leftarrow 0$
```

Example violation

Algorithm Dekker's for T_0

```
1: non-critical section
2: $flag[0] \leftarrow 1$
3: while $flag[1] = 1$ do
4:     if $turn = 1$ then
5:         $flag[0] \leftarrow 0$
6:         await $turn = 0$
7:         $flag[0] \leftarrow 1$
8: critical section
9: $\underline{turn \leftarrow 1}$
10: $flag[0] \leftarrow 0$
```

$flag[0] = 0$

$flag[1] = 0$

$turn = 1$

Algorithm Dekker's for T_1

```
1: non-critical section
2: $flag[1] \leftarrow 1$
3: while $flag[0] = 1$ do
4:     if $turn = 0$ then
5:         $flag[1] \leftarrow 0$
6:         await $\underline{turn = 1}$
7:         $flag[1] \leftarrow 1$
8: critical section
9: $turn \leftarrow 0$
10: $flag[1] \leftarrow 0$
```

Example violation

Is this execution realistic?

Example violation

Is this execution realistic?

It depends on our *memory model*

Example violation

Is this execution realistic?
It depends on our *memory model*

Goal: verify many mutual exclusion algorithms under 6 different memory models

Memory models

Memory models

Memory: only read/write registers

Memory models

Memory: only read/write registers

Memory model

- Operations *blocking* each other
- Register behaviour when operations *overlap*

Memory models

Memory: only read/write registers

Memory model

- Operations *blocking* each other
- Register behaviour when operations *overlap*
- ~~weak memory models, caching, operation reordering, etc.~~

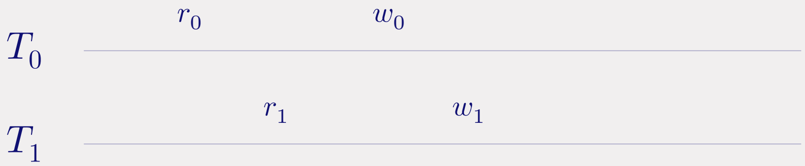
Memory models

Memory: only read/write registers

Memory model

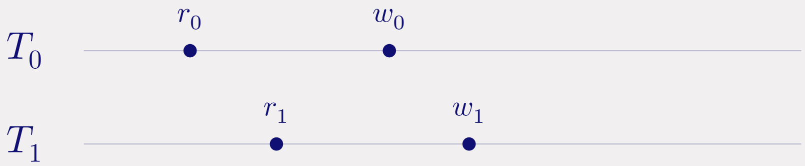
- Operations *blocking* each other
- Register behaviour when operations *overlap*
- ~~weak memory models, caching, operation reordering, etc.~~

Memory models - blocking



Memory models - blocking

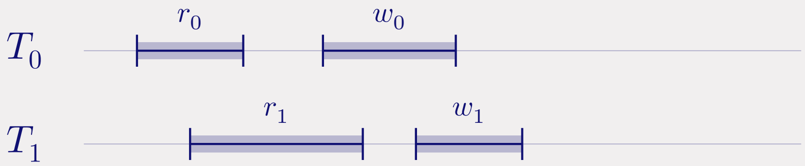
Original problem assumes *atomicity*: operations cannot overlap



Memory models - blocking

Original problem assumes *atomicity*: operations cannot overlap

Operations have *duration*

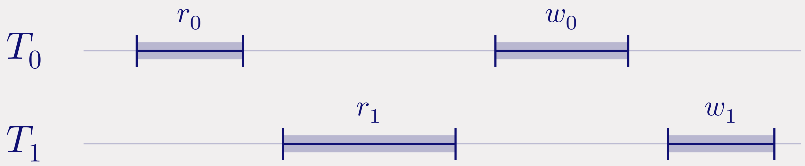


Memory models - blocking

Original problem assumes *atomicity*: operations cannot overlap

Operations have *duration*

Duration + no overlap = operations *block* each other



Memory models - blocking

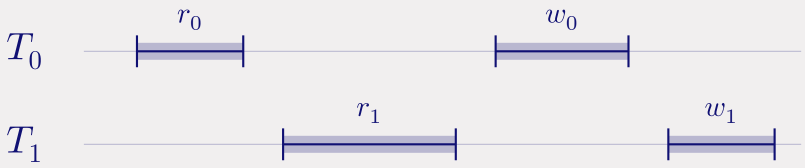
4 reasonable views on blocking

1.
2.
3.
4.

Memory models - blocking

4 reasonable views on blocking

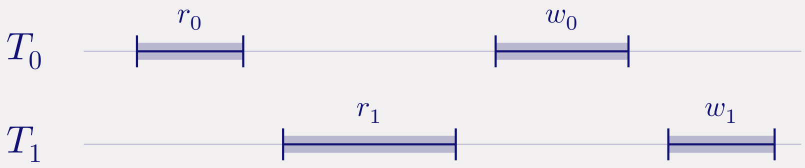
1.
2.
3.
4. Atomicity assumption: no overlap at all



Memory models - blocking

4 reasonable views on blocking

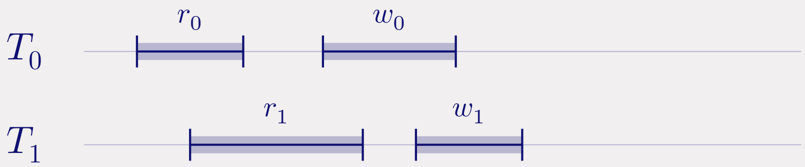
1.
2.
3.
4. Blocking reads and writes



Memory models - blocking

4 reasonable views on blocking

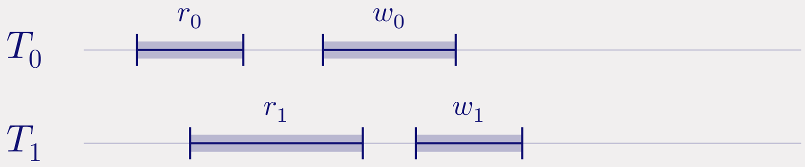
1. Drop the atomicity assumption: all overlap allowed
2.
3.
4. **Blocking reads and writes**



Memory models - blocking

4 reasonable views on blocking

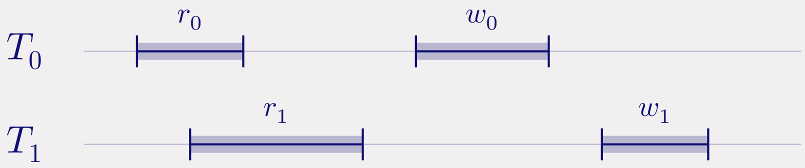
1. **Non-blocking reads and writes**
2.
3.
4. **Blocking reads and writes**



Memory models - blocking

4 reasonable views on blocking

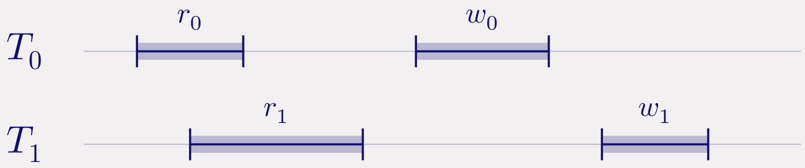
1. **Non-blocking reads and writes**
2. Writes cannot overlap anything $+$ prioritize writes
3. Writes cannot overlap anything
4. **Blocking reads and writes**



Memory models - blocking

4 reasonable views on blocking

1. **Non-blocking reads and writes**
2. **Blocking writes and non-blocking reads**
3. **Blocking model with concurrent reads**
4. **Blocking reads and writes**



Memory models - blocking

4 reasonable views on blocking

1. **Non-blocking reads and writes**
2. **Blocking writes and non-blocking reads**
3. **Blocking model with concurrent reads**
4. **Blocking reads and writes**

Memory models - blocking

4 reasonable views on blocking

1. **Non-blocking reads and writes**
2. **Blocking writes and non-blocking reads**
3. **Blocking model with concurrent reads**
4. **Blocking reads and writes**

	r block r'?	r block w?	w block r?	w block w'?
1	✗	✗	✗	✗
2	✗	✗	✓	✓
3	✗	✓	✓	✓
4	✓	✓	✓	✓

Memory models - overlap

Memory models - overlap

Behaviour in case of overlap

- Based on Lamport's work

safe registers

regular registers

atomic registers

Memory models - overlap

Behaviour in case of overlap

- Based on Lamport’s work
- Only overlapping writes matter

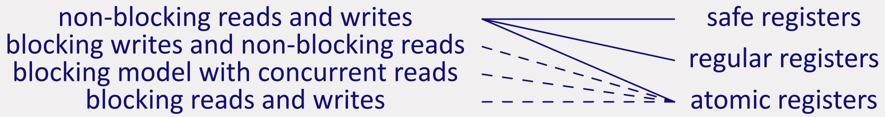
non-blocking reads and writes
blocking writes and non-blocking reads
blocking model with concurrent reads
blocking reads and writes

safe registers
regular registers
atomic registers

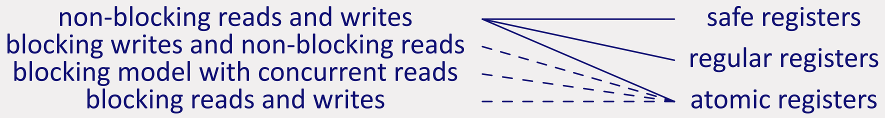
Memory models - overlap

Behaviour in case of overlap

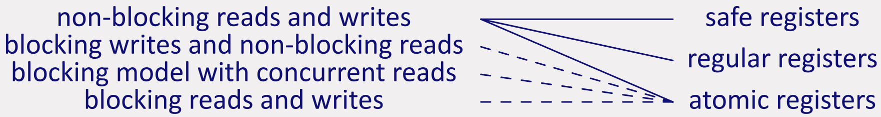
- Based on Lamport's work
- Only overlapping writes matter


non-blocking reads and writes
blocking writes and non-blocking reads
blocking model with concurrent reads
blocking reads and writes
safe registers
regular registers
atomic registers

Modelling memory models

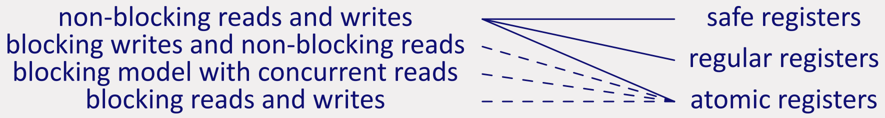

non-blocking reads and writes
blocking writes and non-blocking reads
blocking model with concurrent reads
blocking reads and writes
safe registers
regular registers
atomic registers

Modelling memory models


non-blocking reads and writes
blocking writes and non-blocking reads
blocking model with concurrent reads
blocking reads and writes
safe registers
regular registers
atomic registers

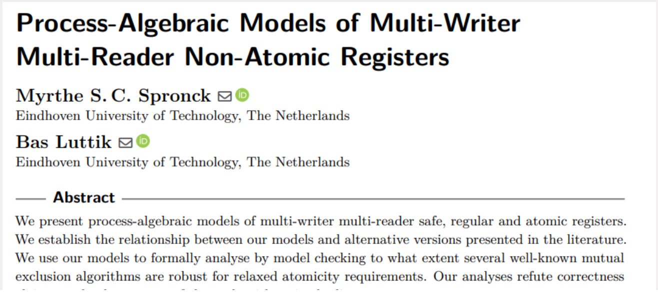
Capture in two dimensions:

Modelling memory models


non-blocking reads and writes
blocking writes and non-blocking reads
blocking model with concurrent reads
blocking reads and writes
safe registers
regular registers
atomic registers

Capture in two dimensions:

- Register types via models



Process-Algebraic Models of Multi-Writer Multi-Reader Non-Atomic Registers

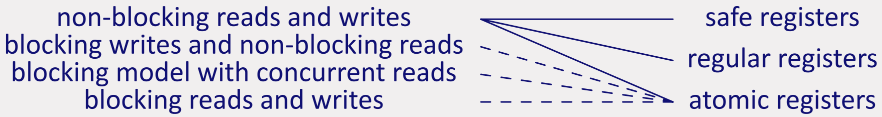
Myrthe S. C. Spronck
Eindhoven University of Technology, The Netherlands

Bas Luttik
Eindhoven University of Technology, The Netherlands

Abstract

We present process-algebraic models of multi-writer multi-reader safe, regular and atomic registers. We establish the relationship between our models and alternative versions presented in the literature. We use our models to formally analyse by model checking to what extent several well-known mutual exclusion algorithms are robust for relaxed atomicity requirements. Our analyses refute correctness

Modelling memory models


non-blocking reads and writes
blocking writes and non-blocking reads
blocking model with concurrent reads
blocking reads and writes
safe registers
regular registers
atomic registers

Capture in two dimensions:

- Register types via models
- Blocking behaviour via formulas

Blocking in formulas

Leverage the *justness* assumption

Completeness criteria: disregard “incomplete” executions

Blocking in formulas

Leverage the *justness* assumption
Completeness criteria: disregard “incomplete” executions

Justness: an enabled event must eventually occur or be “interfered with”

Blocking in formulas

Leverage the *justness* assumption
Completeness criteria: disregard “incomplete” executions

Justness: an enabled event must eventually occur or be “interfered with”

$$\begin{tikzcd}
s_0 \arrow[loop left, "b"] \arrow[rr, "a"] & & s_1
\end{tikzcd}$$

Blocking in formulas

Leverage the *justness* assumption
Completeness criteria: disregard “incomplete” executions

Justness: an enabled event must eventually occur or be “interfered with”

$$\begin{tikzcd}
s_0 \arrow[loop left, "b"] \arrow[r, "a"] & s_1
\end{tikzcd}$$

Depends on a *concurrency relation* $\smile\!\!\bullet$

- Encodes knowledge of real system
- Interference given by $\not\smile\!\!\bullet$

Blocking in formulas

Leverage the *justness* assumption
Completeness criteria: disregard “incomplete” executions

Justness: an enabled event must eventually occur or be “interfered with”

$$\begin{tikzcd} s_0 \arrow[loop left, "b"] \arrow[rr, "a"] & & s_1 \end{tikzcd}$$

Depends on a *concurrency relation* $\smile\!\!\!\bullet$

- Encodes knowledge of real system
- Interference given by $\not\smile\!\!\!\bullet$
- Interference $\approx$ blocking

Blocking via concurrency relation

Blocking via concurrency relation

	r block r'?	r block w?	w block r?	w block w'?
1	✗	✗	✗	✗
2	✗	✗	✓	✓
3	✗	✓	✓	✓
4	✓	✓	✓	✓

Blocking via concurrency relation

	$r' \not\smile\!\!\bullet\ r$?	$w \not\smile\!\!\bullet\ r$?	$r \not\smile\!\!\bullet\ w$?	$w' \not\smile\!\!\bullet\ w$?
1	✗	✗	✗	✗
2	✗	✗	✓	✓
3	✗	✓	✓	✓
4	✓	✓	✓	✓

Blocking via concurrency relation

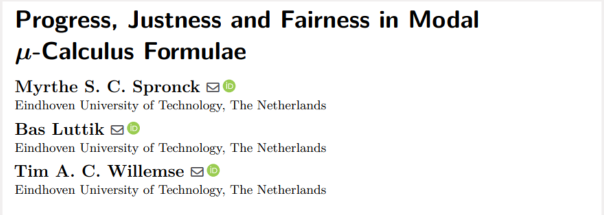
	$r' \not\smile\!\bullet\, r$?	$w \not\smile\!\bullet\, r$?	$r \not\smile\!\bullet\, w$?	$w' \not\smile\!\bullet\, w$?
1	✗	✗	✗	✗
2	✗	✗	✓	✓
3	✗	✓	✓	✓
4	✓	✓	✓	✓

Incorporate justness + concurrency relations into formulas

Blocking via concurrency relation

	$r' \not\smile\!\!\bullet\ r?$	$w \not\smile\!\!\bullet\ r?$	$r \not\smile\!\!\bullet\ w?$	$w' \not\smile\!\!\bullet\ w?$
1	✗	✗	✗	✗
2	✗	✗	✓	✓
3	✗	✓	✓	✓
4	✓	✓	✓	✓

Incorporate justness + concurrency relations into formulas


Progress, Justness and Fairness in Modal μ-Calculus Formulae

Myrthe S. C. Spronck
Eindhoven University of Technology, The Netherlands

Bas Luttik
Eindhoven University of Technology, The Netherlands

Tim A. C. Willemse
Eindhoven University of Technology, The Netherlands

Results - Dekker’s algorithm

Results - Dekker’s algorithm

Algorithm	*# threads*	*Safe*	*Regular*	*Atomic*			
		$\smile\!\bullet_1$	$\smile\!\bullet_1$	$\smile\!\bullet_1$	$\smile\!\bullet_2$	$\smile\!\bullet_3$	$\smile\!\bullet_4$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
Dekker	2	M	M	S	D/S	M	M
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$

Results - Dekker’s algorithm

Algorithm	*# threads*	*Safe*	*Regular*	*Atomic*			
		$\smile\!\!\bullet_1$	$\smile\!\!\bullet_1$	$\smile\!\!\bullet_1$	$\smile\!\!\bullet_2$	$\smile\!\!\bullet_3$	$\smile\!\!\bullet_4$
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
Dekker	2	M	M	S	D/S	M	M
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮

Starvation freedom counterexample

- Recall: repeated writes prevent read

Results - Dekker's algorithm

Algorithm	*# threads*	*Safe*	*Regular*	*Atomic*			
		$\curvearrowright_1$	$\curvearrowright_1$	$\curvearrowright_1$	$\curvearrowright_2$	$\curvearrowright_3$	$\curvearrowright_4$
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
Dekker	2	M	M	S	$\boxed{\text{D/S}}$	M	M
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮

Starvation freedom counterexample

- Recall: repeated writes prevent read
- Easily fixed

Results - full

Algorithm	*# threads*	*Safe*	*Regular*	*Atomic*			
		$\smile\!\bullet_1$	$\smile\!\bullet_1$	$\smile\!\bullet_1$	$\smile\!\bullet_2$	$\smile\!\bullet_3$	$\smile\!\bullet_4$
Anderson	2	S	S	S	S	M	M
Aravind BLRU	3	S	S	S	M/S	M	M
Attiya-Welch (orig.)	2	D/S	S	S	D	M	M
Attiya-Welch (var.)	2	M/S	M/S	S	D	M	M
Burns-Lynch	3	D	D	D	D	M	M
Dekker	2	M	M	S	D/S	M	M
Dekker RW-safe	2	S	S	S	D	M	M
Dekker RW-safe (DFtoSF)	2	S	S	S	S	M	M
Dijkstra	3	M	D	D	M	M	M
Kessels	2	X	X	S	S	M	M
Knuth	3	M	S	S	M	M	M
Lamport 1-bit	3	D	D	D	D	M	M
Lamport 1-bit (DFtoSF)	3	S	S	S	S	M	M
Lamport 3-bit	3	S	S	S	S	M	M
Peterson	2	X	X	S	S	M	M
Szymanski flag (int.)	3	X	X	S	S	M	M
Szymanski flag (bit)	3	X	X	X	X	X	X
Szymanski 3-bit lin. wait	3/2	X/S	X/S	X/S	X/S	X/M	X/M

Conclusion

Conclusion

Contributions

- 4 blocking behaviours captured
- Many mutual exclusion algorithms verified
  - 3 properties
  - 6 memory models

Conclusion

Contributions

- 4 blocking behaviours captured
- Many mutual exclusion algorithms verified
    - 3 properties
    - 6 memory models

Future work

- Other properties, e.g. bounded bypass
- Impact of busy waiting
- Arbitrary numbers of threads?

Just Verification of Mutual Exclusion Algorithms

Rob van Glabbeek, Bas Luttik, and Myrthe Spronck

28 August 2025

(Im)possibility of liveness

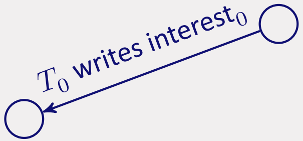
Starvation freedom is *impossible* with I and A

- Reads interfere with writes
- Observation made previously
- In short: repeated reads prevent communicating interest

(Im)possibility of liveness

Starvation freedom is *impossible* with I and A

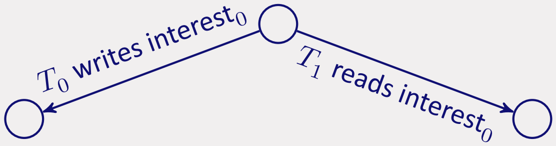
- Reads interfere with writes
- Observation made previously
- In short: repeated reads prevent communicating interest



(Im)possibility of liveness

Starvation freedom is *impossible* with I and A

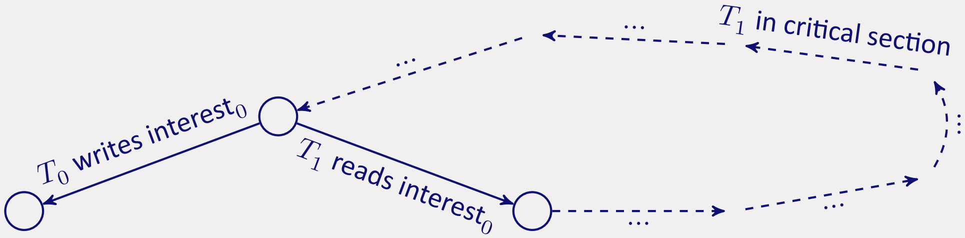
- Reads interfere with writes
- Observation made previously
- In short: repeated reads prevent communicating interest


T_0 writes interest_0
T_1 reads interest_0

(Im)possibility of liveness

Starvation freedom is *impossible* with I and A

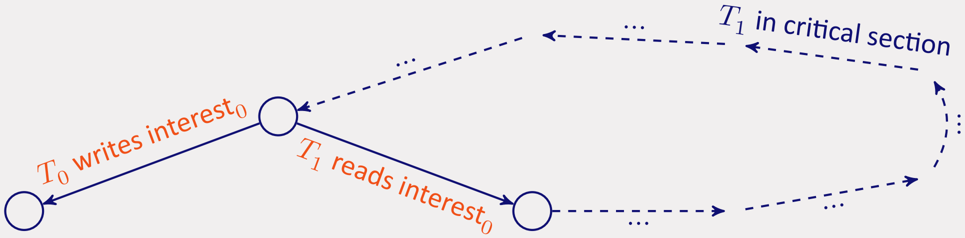
- Reads interfere with writes
- Observation made previously
- In short: repeated reads prevent communicating interest



(Im)possibility of liveness

Starvation freedom is *impossible* with I and A

- Reads interfere with writes
- Observation made previously
- In short: repeated reads prevent communicating interest



(Im)possibility of liveness

Deadlock freedom *always observed to be violated* with I and A

- Might not be impossible
- Busy waiting often to blame

(Im)possibility of liveness

Deadlock freedom *always observed to be violated* with I and A

- Might not be impossible
- Busy waiting often to blame
- Both stuck in entry or one stuck in exit

Algorithm Peterson's algorithm

```
1: $\mathit{flag}[i] \leftarrow \mathit{true}$
2: $\mathit{turn} \leftarrow i$
3: await $\mathit{flag}[j] = \mathit{false} \vee \mathit{turn} = j$
4: critical section
5: $\mathit{flag}[i] \leftarrow \mathit{false}$
```

(Im)possibility of liveness

Deadlock freedom *always observed to be violated* with I and A

- Might not be impossible
- Busy waiting often to blame
- Both stuck in entry or one stuck in exit

Algorithm Peterson's algorithm

```
1: $\mathit{flag}[i] \leftarrow \mathit{true}$
2: $\underline{\mathit{turn} \leftarrow i}$
3: **await** $\mathit{flag}[j] = \mathit{false} \lor \underline{\mathit{turn} = j}$
4: **critical section**
5: $\mathit{flag}[i] \leftarrow \mathit{false}$
```

(Im)possibility of liveness

Deadlock freedom *always observed to be violated* with I and A

- Might not be impossible
- Busy waiting often to blame
- Both stuck in entry or one stuck in exit

Algorithm Peterson's algorithm

```
1: $\mathit{flag}[i] \leftarrow \mathit{true}$
2: $\mathit{turn} \leftarrow i$
3: **await** $\underline{\mathit{flag}[j] = \mathit{false}} \lor \mathit{turn} = j$
4: **critical section**
5: $\underline{\mathit{flag}[i] \leftarrow \mathit{false}}$
```

An interesting violation

- Dekker with safe-T: M

An interesting violation

- Dekker with safe-T: M

Algorithm Dekker's for T_0

```
1: non-critical section
2: $flag[0] \leftarrow 1$
3: while $flag[1] = 1$ do
4:     if $turn = 1$ then
5:         $flag[0] \leftarrow 0$
6:         await $turn = 0$
7:         $flag[0] \leftarrow 1$
8: critical section
9: $turn \leftarrow 1$
10: $flag[0] \leftarrow 0$
```

$flag[0] = 0$

$flag[1] = 0$

$turn = 0$

Algorithm Dekker's for T_1

```
1: non-critical section
2: $flag[1] \leftarrow 1$
3: while $flag[0] = 1$ do
4:     if $turn = 0$ then
5:         $flag[1] \leftarrow 0$
6:         await $turn = 1$
7:         $flag[1] \leftarrow 1$
8: critical section
9: $turn \leftarrow 0$
10: $flag[1] \leftarrow 0$
```

An interesting violation

- Dekker with safe-T: M

Algorithm Dekker's for T_0

```
 1: non-critical section
 2: $flag[0] \leftarrow 1$
 3: while $flag[1] = 1$ do
 4:     if $turn = 1$ then
 5:         $flag[0] \leftarrow 0$
 6:         await $turn = 0$
 7:         $flag[0] \leftarrow 1$
 8: critical section
 9: $turn \leftarrow 1$
10: $flag[0] \leftarrow 0$
```

$flag[0] = 1$

$flag[1] = 0$

$turn = 0$

Algorithm Dekker's for T_1

```
 1: non-critical section
 2: $flag[1] \leftarrow 1$
 3: while $flag[0] = 1$ do
 4:     if $turn = 0$ then
 5:         $flag[1] \leftarrow 0$
 6:         await $turn = 1$
 7:         $flag[1] \leftarrow 1$
 8: critical section
 9: $turn \leftarrow 0$
10: $flag[1] \leftarrow 0$
```

An interesting violation

- Dekker with safe-T: M

```
Algorithm Dekker's for $T_0$
1:  non-critical section
2:  $flag[0] \leftarrow 1$
3:  while $flag[1] = 1$ do
4:      if $turn = 1$ then
5:          $flag[0] \leftarrow 0$
6:          await $turn = 0$
7:          $flag[0] \leftarrow 1$
8:  critical section
9:  $turn \leftarrow 1$
10: $flag[0] \leftarrow 0$
```

$flag[0] = 1$

$flag[1] = 0$

$turn = 0$

```
Algorithm Dekker's for $T_1$
1:  non-critical section
2:  $flag[1] \leftarrow 1$
3:  while $flag[0] = 1$ do
4:      if $turn = 0$ then
5:          $flag[1] \leftarrow 0$
6:          await $turn = 1$
7:          $flag[1] \leftarrow 1$
8:  critical section
9:  $turn \leftarrow 0$
10: $flag[1] \leftarrow 0$
```

An interesting violation

- Dekker with safe-T: M

Algorithm Dekker's for T_0

```
1: non-critical section
2: $flag[0] \leftarrow 1$
3: while $flag[1] = 1$ do
4:     if $turn = 1$ then
5:         $flag[0] \leftarrow 0$
6:         await $turn = 0$
7:         $flag[0] \leftarrow 1$
8: critical section
9: $turn \leftarrow 1$
10: $flag[0] \leftarrow 0$
```

$flag[0] = 1$

$flag[1] = 0$

$turn = 0$

Algorithm Dekker's for T_1

```
1: non-critical section
2: $flag[1] \leftarrow 1$
3: while $flag[0] = 1$ do
4:     if $turn = 0$ then
5:         $flag[1] \leftarrow 0$
6:         await $turn = 1$
7:         $flag[1] \leftarrow 1$
8: critical section
9: $turn \leftarrow 0$
10: $flag[1] \leftarrow 0$
```

An interesting violation

- Dekker with safe-T: M

Algorithm Dekker's for T_0

```
1:  non-critical section
2:  $flag[0] \leftarrow 1$
3:  while $flag[1] = 1$ do
4:      if $turn = 1$ then
5:          $flag[0] \leftarrow 0$
6:          await $turn = 0$
7:          $flag[0] \leftarrow 1$
8:  critical section
9:  $turn \leftarrow 1$
10: $flag[0] \leftarrow 0$
```

$flag[0] = 1$

$flag[1] = 0$

$turn = 1$

Algorithm Dekker's for T_1

```
1:  non-critical section
2:  $flag[1] \leftarrow 1$
3:  while $flag[0] = 1$ do
4:      if $turn = 0$ then
5:          $flag[1] \leftarrow 0$
6:          await $turn = 1$
7:          $flag[1] \leftarrow 1$
8:  critical section
9:  $turn \leftarrow 0$
10: $flag[1] \leftarrow 0$
```

An interesting violation

- Dekker with safe-T: M

Algorithm Dekker's for T_0

```
1: non-critical section
2: $flag[0] \leftarrow 1$
3: while $flag[1] = 1$ do
4:     if $turn = 1$ then
5:         $flag[0] \leftarrow 0$
6:         await $turn = 0$
7:         $flag[0] \leftarrow 1$
8: critical section
9: $turn \leftarrow 1$
10: $flag[0] \leftarrow 0$
```

$flag[0] = ?$

$flag[1] = 0$

$turn = 1$

new-old inversion

old $= 1$

new $= 0$

Algorithm Dekker's for T_1

```
1: non-critical section
2: $flag[1] \leftarrow 1$
3: while $flag[0] = 1$ do
4:     if $turn = 0$ then
5:         $flag[1] \leftarrow 0$
6:         await $turn = 1$
7:         $flag[1] \leftarrow 1$
8: critical section
9: $turn \leftarrow 0$
10: $flag[1] \leftarrow 0$
```

An interesting violation

- Dekker with safe-T: M

Algorithm Dekker's for T_0

```
1: non-critical section
2: $flag[0] \leftarrow 1$
3: while $flag[1] = 1$ do
4:     if $turn = 1$ then
5:         $flag[0] \leftarrow 0$
6:         await $turn = 0$
7:         $flag[0] \leftarrow 1$
8: critical section
9: $turn \leftarrow 1$
10: $\underline{flag[0] \leftarrow 0}$
```

$flag[0] = ?$

$flag[1] = 1$

$turn = 1$

new-old inversion

old $= 1$

new $= 0$

Algorithm Dekker's for T_1

```
1: non-critical section
2: $\underline{flag[1] \leftarrow 1}$
3: while $flag[0] = 1$ do
4:     if $turn = 0$ then
5:         $flag[1] \leftarrow 0$
6:         await $turn = 1$
7:         $flag[1] \leftarrow 1$
8: critical section
9: $turn \leftarrow 0$
10: $flag[1] \leftarrow 0$
```

An interesting violation

- Dekker with safe-T: M

Algorithm Dekker's for T_0

```
1: non-critical section
2: $flag[0] \leftarrow 1$
3: while $flag[1] = 1$ do
4:     if $turn = 1$ then
5:         $flag[0] \leftarrow 0$
6:         await $turn = 0$
7:         $flag[0] \leftarrow 1$
8: critical section
9: $turn \leftarrow 1$
10: $\underline{flag[0] \leftarrow 0}$
```

$flag[0] = ?$

$flag[1] = 1$

$turn = 1$

new-old inversion

old $= 1$

new $= 0$

Algorithm Dekker's for T_1

```
1: non-critical section
2: $flag[1] \leftarrow 1$
3: while $\underline{flag[0] = 1}$ do
4:     if $turn = 0$ then
5:         $flag[1] \leftarrow 0$
6:         await $turn = 1$
7:         $flag[1] \leftarrow 1$
8: critical section
9: $turn \leftarrow 0$
10: $flag[1] \leftarrow 0$
```

An interesting violation

- Dekker with safe-T: M

```
Algorithm Dekker's for $T_0$
 1: non-critical section
 2: $flag[0] \leftarrow 1$
 3: while $flag[1] = 1$ do
 4:     if $turn = 1$ then
 5:         $flag[0] \leftarrow 0$
 6:         await $turn = 0$
 7:         $flag[0] \leftarrow 1$
 8: critical section
 9: $turn \leftarrow 1$
10: $flag[0] \leftarrow 0$
```

$flag[0] = ?$

$flag[1] = 1$

$turn = 1$

new-old inversion

old $= 1$

new $= 0$

```
Algorithm Dekker's for $T_1$
 1: non-critical section
 2: $flag[1] \leftarrow 1$
 3: while $flag[0] = 1$ do
 4:     if $turn = 0$ then
 5:         $flag[1] \leftarrow 0$
 6:         await $turn = 1$
 7:         $flag[1] \leftarrow 1$
 8: critical section
 9: $turn \leftarrow 0$
10: $flag[1] \leftarrow 0$
```

An interesting violation

- Dekker with safe-T: M

```
Algorithm Dekker's for $T_0$
 1: non-critical section
 2: $flag[0] \leftarrow 1$
 3: while $flag[1] = 1$ do
 4:     if $turn = 1$ then
 5:         $flag[0] \leftarrow 0$
 6:         await $turn = 0$
 7:         $flag[0] \leftarrow 1$
 8: critical section
 9: $turn \leftarrow 1$
10: $\underline{flag[0] \leftarrow 0}$
```

$flag[0] = ?$

$flag[1] = 1$

$turn = 0$

new-old inversion

old $= 1$

new $= 0$

```
Algorithm Dekker's for $T_1$
 1: non-critical section
 2: $flag[1] \leftarrow 1$
 3: while $flag[0] = 1$ do
 4:     if $turn = 0$ then
 5:         $flag[1] \leftarrow 0$
 6:         await $turn = 1$
 7:         $flag[1] \leftarrow 1$
 8: critical section
 9: $\underline{turn \leftarrow 0}$
10: $flag[1] \leftarrow 0$
```

An interesting violation

- Dekker with safe-T: M

Algorithm Dekker's for T_0

```
1: non-critical section
2: $flag[0] \leftarrow 1$
3: while $flag[1] = 1$ do
4:     if $turn = 1$ then
5:         $flag[0] \leftarrow 0$
6:         await $turn = 0$
7:         $flag[0] \leftarrow 1$
8: critical section
9: $turn \leftarrow 1$
10: $flag[0] \leftarrow 0$
```

$flag[0] = ?$

$flag[1] = 0$

$turn = 0$

new-old inversion

old $= 1$

new $= 0$

Algorithm Dekker's for T_1

```
1: non-critical section
2: $flag[1] \leftarrow 1$
3: while $flag[0] = 1$ do
4:     if $turn = 0$ then
5:         $flag[1] \leftarrow 0$
6:         await $turn = 1$
7:         $flag[1] \leftarrow 1$
8: critical section
9: $turn \leftarrow 0$
10: $flag[1] \leftarrow 0$
```

An interesting violation

- Dekker with safe-T: M

Algorithm Dekker's for T_0

```
1: non-critical section
2: $flag[0] \leftarrow 1$
3: while $flag[1] = 1$ do
4:     if $turn = 1$ then
5:         $flag[0] \leftarrow 0$
6:         await $turn = 0$
7:         $flag[0] \leftarrow 1$
8: critical section
9: $turn \leftarrow 1$
10: $flag[0] \leftarrow 0$
```

$flag[0] = ?$

$flag[1] = 0$

$turn = 0$

new-old inversion

old $= 1$

new $= 0$

Algorithm Dekker's for T_1

```
1: non-critical section
2: $flag[1] \leftarrow 1$
3: while $flag[0] = 1$ do
4:     if $turn = 0$ then
5:         $flag[1] \leftarrow 0$
6:         await $turn = 1$
7:         $flag[1] \leftarrow 1$
8: critical section
9: $turn \leftarrow 0$
10: $flag[1] \leftarrow 0$
```

An interesting violation

- Dekker with safe-T: M

Algorithm Dekker's for T_0

```
 1: non-critical section
 2: $flag[0] \leftarrow 1$
 3: while $flag[1] = 1$ do
 4:     if $turn = 1$ then
 5:         $flag[0] \leftarrow 0$
 6:         await $turn = 0$
 7:         $flag[0] \leftarrow 1$
 8: critical section
 9: $turn \leftarrow 1$
10: $flag[0] \leftarrow 0$
```

$flag[0] = ?$

$flag[1] = 1$

$turn = 0$

new-old inversion

old $= 1$

new $= 0$

Algorithm Dekker's for T_1

```
 1: non-critical section
 2: $flag[1] \leftarrow 1$
 3: while $flag[0] = 1$ do
 4:     if $turn = 0$ then
 5:         $flag[1] \leftarrow 0$
 6:         await $turn = 1$
 7:         $flag[1] \leftarrow 1$
 8: critical section
 9: $turn \leftarrow 0$
10: $flag[1] \leftarrow 0$
```

An interesting violation

- Dekker with safe-T: M

Algorithm Dekker's for T_0

```
1: non-critical section
2: $flag[0] \leftarrow 1$
3: while $flag[1] = 1$ do
4:     if $turn = 1$ then
5:         $flag[0] \leftarrow 0$
6:         await $turn = 0$
7:         $flag[0] \leftarrow 1$
8: critical section
9: $turn \leftarrow 1$
10: $\underline{flag[0] \leftarrow 0}$
```

$flag[0] = ?$

$flag[1] = 1$

$turn = 0$

new-old inversion

old $= 1$

new $= 0$

Algorithm Dekker's for T_1

```
1: non-critical section
2: $flag[1] \leftarrow 1$
3: while $\underline{flag[0] = 1}$ do
4:     if $turn = 0$ then
5:         $flag[1] \leftarrow 0$
6:         await $turn = 1$
7:         $flag[1] \leftarrow 1$
8: critical section
9: $turn \leftarrow 0$
10: $flag[1] \leftarrow 0$
```

An interesting violation

- Dekker with safe-T: M

Algorithm Dekker's for T_0

```
1: non-critical section
2: $flag[0] \leftarrow 1$
3: while $flag[1] = 1$ do
4:     if $turn = 1$ then
5:         $flag[0] \leftarrow 0$
6:         await $turn = 0$
7:         $flag[0] \leftarrow 1$
8: critical section
9: $turn \leftarrow 1$
10: $\underline{flag[0] \leftarrow 0}$
```

$flag[0] = ?$

$flag[1] = 1$

$turn = 0$

new-old inversion

old $= 1$

new $= 0$

Algorithm Dekker's for T_1

```
1: non-critical section
2: $flag[1] \leftarrow 1$
3: while $flag[0] = 1$ do
4:     if $\underline{turn = 0}$ then
5:         $flag[1] \leftarrow 0$
6:         await $turn = 1$
7:         $flag[1] \leftarrow 1$
8: critical section
9: $turn \leftarrow 0$
10: $flag[1] \leftarrow 0$
```

An interesting violation

- Dekker with safe-T: M

Algorithm Dekker's for T_0

```
1: non-critical section
2: $flag[0] \leftarrow 1$
3: while $flag[1] = 1$ do
4:     if $turn = 1$ then
5:         $flag[0] \leftarrow 0$
6:         await $turn = 0$
7:         $flag[0] \leftarrow 1$
8: critical section
9: $turn \leftarrow 1$
10: $\underline{flag[0] \leftarrow 0}$
```

$flag[0] = ?$

$flag[1] = 0$

$turn = 0$

new-old inversion

old $= 1$

new $= 0$

Algorithm Dekker's for T_1

```
1: non-critical section
2: $flag[1] \leftarrow 1$
3: while $flag[0] = 1$ do
4:     if $turn = 0$ then
5:         $\underline{flag[1] \leftarrow 0}$
6:         await $turn = 1$
7:         $flag[1] \leftarrow 1$
8: critical section
9: $turn \leftarrow 0$
10: $flag[1] \leftarrow 0$
```

An interesting violation

- Dekker with safe-T: M

Algorithm Dekker's for T_0

```
1:  non-critical section
2:  $flag[0] \leftarrow 1$
3:  while $flag[1] = 1$ do
4:      if $turn = 1$ then
5:          $flag[0] \leftarrow 0$
6:          await $turn = 0$
7:          $flag[0] \leftarrow 1$
8:  critical section
9:  $turn \leftarrow 1$
10: $\underline{flag[0] \leftarrow 0}$
```

$flag[0] = ?$

$flag[1] = 0$

$turn = 0$

new-old inversion

old $= 1$

new $= 0$

Algorithm Dekker's for T_1

```
1:  non-critical section
2:  $flag[1] \leftarrow 1$
3:  while $flag[0] = 1$ do
4:      if $turn = 0$ then
5:          $flag[1] \leftarrow 0$
6:          await $\underline{turn = 1}$
7:          $flag[1] \leftarrow 1$
8:  critical section
9:  $turn \leftarrow 0$
10: $flag[1] \leftarrow 0$
```

An interesting violation

- Dekker with safe-T: M

Algorithm Dekker's for T_0

```
1: non-critical section
2: $flag[0] \leftarrow 1$
3: while $flag[1] = 1$ do
4:     if $turn = 1$ then
5:         $flag[0] \leftarrow 0$
6:         await $turn = 0$
7:         $flag[0] \leftarrow 1$
8: critical section
9: $turn \leftarrow 1$
10: $\underline{flag[0] \leftarrow 0}$
```

$flag[0] = 0$

$flag[1] = 0$

$turn = 0$

new-old inversion

old $= 1$

new $= 0$

Algorithm Dekker's for T_1

```
1: non-critical section
2: $flag[1] \leftarrow 1$
3: while $flag[0] = 1$ do
4:     if $turn = 0$ then
5:         $flag[1] \leftarrow 0$
6:         await $\underline{turn = 1}$
7:         $flag[1] \leftarrow 1$
8: critical section
9: $turn \leftarrow 0$
10: $flag[1] \leftarrow 0$
```

An interesting violation

- Dekker with safe-T: M

```
Algorithm Dekker's for $T_0$
1: non-critical section
2: $flag[0] \leftarrow 1$
3: while $flag[1] = 1$ do
4:     if $turn = 1$ then
5:         $flag[0] \leftarrow 0$
6:         await $turn = 0$
7:         $flag[0] \leftarrow 1$
8: critical section
9: $turn \leftarrow 1$
10: $flag[0] \leftarrow 0$
```

$flag[0] = 0$

$flag[1] = 0$

$turn = 0$

new-old inversion

old $= 1$

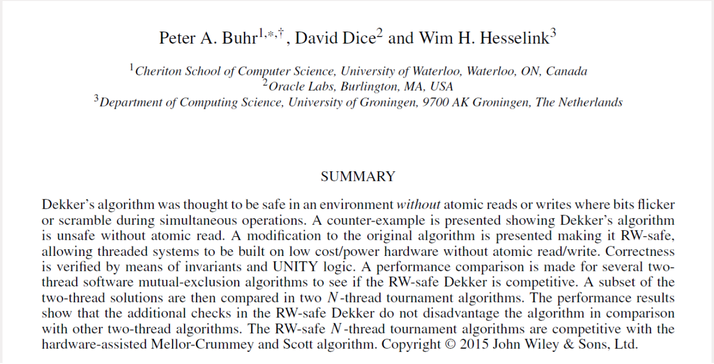
new $= 0$

infinite non-critical

```
Algorithm Dekker's for $T_1$
1: non-critical section
2: $flag[1] \leftarrow 1$
3: while $flag[0] = 1$ do
4:     if $turn = 0$ then
5:         $flag[1] \leftarrow 0$
6:         await $turn = 1$
7:         $flag[1] \leftarrow 1$
8: critical section
9: $turn \leftarrow 0$
10: $flag[1] \leftarrow 0$
```

An interesting violation

- Dekker with safe-T: M
- RW-safe variant: S



Peter A. Buhr[1,*,†], David Dice[2] and Wim H. Hesselink[3]

[1]*Cheriton School of Computer Science, University of Waterloo, Waterloo, ON, Canada*
[2]*Oracle Labs, Burlington, MA, USA*
[3]*Department of Computing Science, University of Groningen, 9700 AK Groningen, The Netherlands*

SUMMARY

Dekker's algorithm was thought to be safe in an environment *without* atomic reads or writes where bits flicker or scramble during simultaneous operations. A counter-example is presented showing Dekker's algorithm is unsafe without atomic read. A modification to the original algorithm is presented making it RW-safe, allowing threaded systems to be built on low cost/power hardware without atomic read/write. Correctness is verified by means of invariants and UNITY logic. A performance comparison is made for several two-thread software mutual-exclusion algorithms to see if the RW-safe Dekker is competitive. A subset of the two-thread solutions are then compared in two N-thread tournament algorithms. The performance results show that the additional checks in the RW-safe Dekker do not disadvantage the algorithm in comparison with other two-thread algorithms. The RW-safe N-thread tournament algorithms are competitive with the hardware-assisted Mellor-Crummey and Scott algorithm. Copyright © 2015 John Wiley & Sons, Ltd.

Just Verification of Mutual Exclusion Algorithms

Rob van Glabbeek, Bas Luttik, and Myrthe Spronck

28 August 2025

Eindhoven University of Technology - Formal System Analysis