Verifying shared-memory mutual exclusion algorithms with non-atomic reads and writes

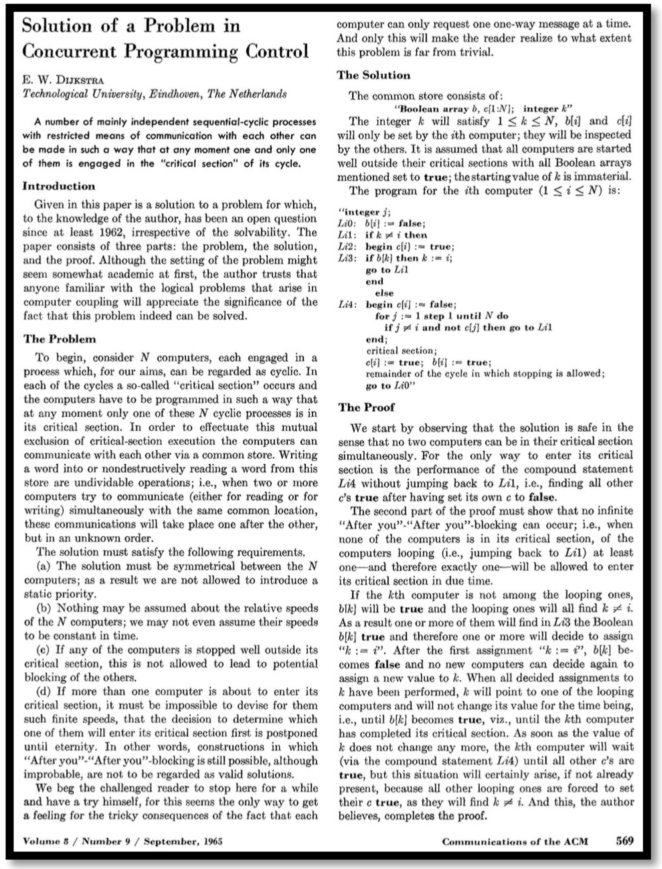
Myrthe Spronck and Bas Luttik

OPCT (June 26, 2023)

MUTUAL EXCLUSION

Edsger W. Dijkstra (1972)

- *critical section*: part of thread code in which some shared resource is accessed
- *mutual exclusion*: at all times, at most one thread has access to the resource
- Goal: insert code before and after critical section to ensure mutual exclusion.



Solution of a Problem in Concurrent Programming Control

E. W. Dijkstra
Technological University, Eindhoven, The Netherlands

A number of mainly independent sequential-cyclic processes with restricted means of communication with each other can be made in such a way that at any moment one and only one of them is engaged in the "critical section" of its cycle.

Introduction

Given in this paper is a solution to a problem for which, to the knowledge of the author, has been an open question since at least 1962, irrespective of the solvability. The paper consists of three parts: the problem, the solution, and the proof. Although the setting of the problem might seem somewhat academic at first, the author trusts that anyone familiar with the logical problems that arise in computer coupling will appreciate the significance of the fact that this problem indeed can be solved.

The Problem

To begin, consider N computers, each engaged in a process which, for our aims, can be regarded as cyclic. In each of the cycles a so-called "critical section" occurs and the computers have to be programmed in such a way that at any moment only one of these N cyclic processes is in its critical section. In order to effectuate this mutual exclusion of critical-section execution the computers can communicate with each other via a common store. Writing a word into or nondestructively reading a word from this store are undividable operations; i.e., when two or more computers try to communicate (either for reading or for writing) simultaneously with the same common location, these communications will take place one after the other, but in an unknown order.

The solution must satisfy the following requirements.

(a) The solution must be symmetrical between the N computers; as a result we are not allowed to introduce a static priority.

(b) Nothing may be assumed about the relative speeds of the N computers; we may not even assume their speeds to be constant in time.

(c) If any of the computers is stopped well outside its critical section, this is not allowed to lead to potential blocking of the others.

(d) If more than one computer is about to enter its critical section, it must be impossible to devise for them such finite speeds, that the decision to determine which one of them will enter its critical section first is postponed until eternity. In other words, constructions in which "After you"-"After you"-blocking is still possible, although improbable, are not to be regarded as valid solutions.

We beg the challenged reader to stop here for a while and have a try himself, for this seems the only way to get a feeling for the tricky consequences of the fact that each computer can only request one one-way message at a time. And only this will make the reader realize to what extent this problem is far from trivial.

The Solution

The common store consists of:

"**Boolean array** b, $c[1:N]$; **integer** k"

The integer k will satisfy $1 \leq k \leq N$, $b[i]$ and $c[i]$ will only be set by the ith computer; they will be inspected by the others. It is assumed that all computers are started well outside their critical sections with all Boolean arrays mentioned set to **true**; the starting value of k is immaterial.

The program for the ith computer ($1 \leq i \leq N$) is:

```
"integer j;
Li0: b[i] := false;
Li1: if k ≠ i then
Li2: begin c[i] := true;
Li3: if b[k] then k := i;
     go to Li1
     end
       else
Li4: begin c[i] := false;
       for j := 1 step 1 until N do
         if j ≠ i and not c[j] then go to Li1
     end;
     critical section;
     c[i] := true;  b[i] := true;
     remainder of the cycle in which stopping is allowed;
     go to Li0"
```

The Proof

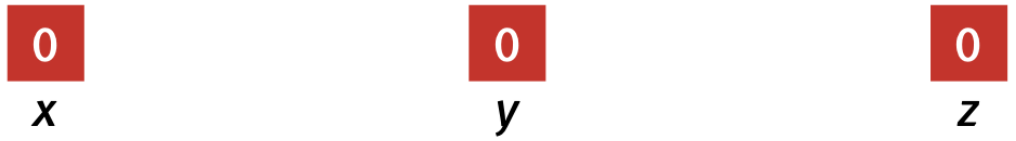
We start by observing that the solution is safe in the sense that no two computers can be in their critical section simultaneously. For the only way to enter its critical section is the performance of the compound statement $Li4$ without jumping back to $Li1$, i.e., finding all other c's **true** after having set its own c to **false**.

The second part of the proof must show that no infinite "After you"-"After you"-blocking can occur; i.e., when none of the computers is in its critical section, of the computers looping (i.e., jumping back to $Li1$) at least one—and therefore exactly one—will be allowed to enter its critical section in due time.

If the kth computer is not among the looping ones, $b[k]$ will be **true** and the looping ones will all find $k \neq i$. As a result one or more of them will find in $Li3$ the Boolean $b[k]$ **true** and therefore one or more will decide to assign "$k := i$". After the first assignment "$k := i$", $b[k]$ becomes **false** and no new computers can decide again to assign a new value to k. When all decided assignments to k have been performed, k will point to one of the looping computers and will not change its value for the time being, i.e., until $b[k]$ becomes **true**, viz., until the kth computer has completed its critical section. As soon as the value of k does not change any more, the kth computer will wait (via the compound statement $Li4$) until all other c's are **true**, but this situation will certainly arise, if not already present, because all other looping ones are forced to set their c **true**, as they will find $k \neq i$. And this, the author believes, completes the proof.

Volume 8 / Number 9 / September, 1965 Communications of the ACM 569

Communications of the ACM 8:9, p. 569, 1965.



The algorithm run by thread $i \in \{1, 2\}$

```
1: ‘noncritical section’
2: $x := i$
3: if $y = 0$ or $y = i$ then skip else goto 2
4: $z := i$
5: if $x = i$ then skip else goto 2
6: $y := i$
7: if $z = i$ then skip else goto 2
8: ‘critical section’
```

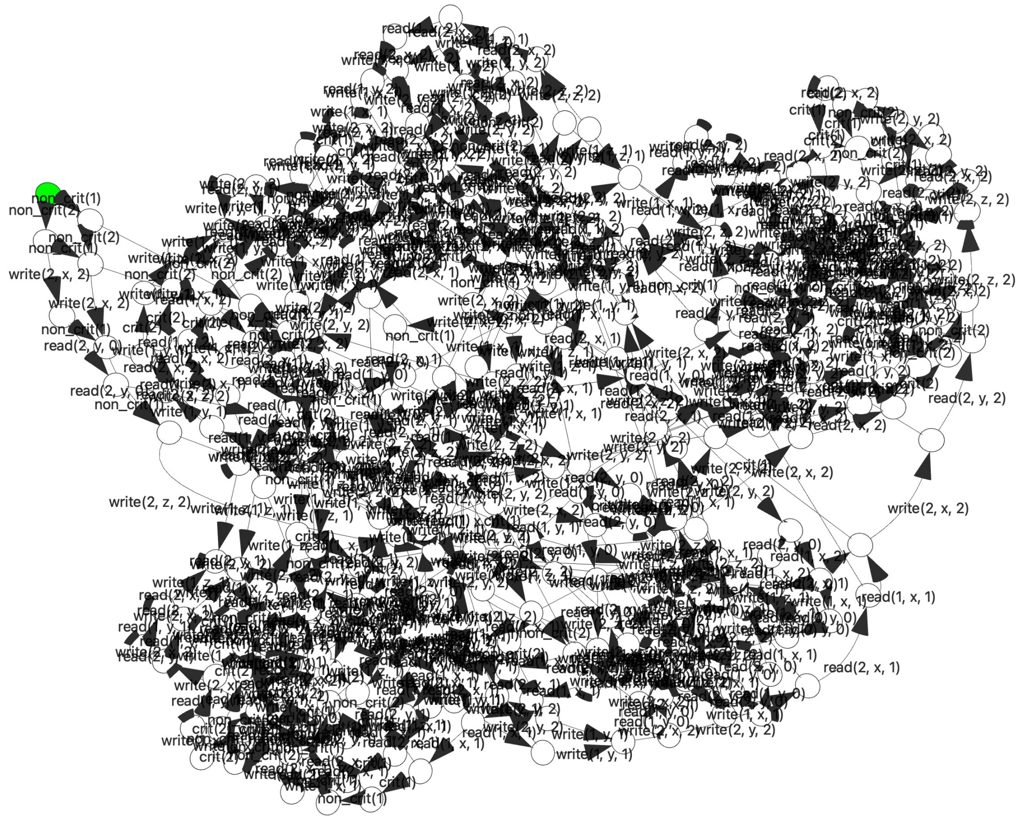
Courtesy of Gerard J. Holzmann. *The SPIN Model Checker – primer and reference manual.* Addison-Wesley, 2003.

FLAWED ALGORITHM

0 0 0
x y z

The algorithm run by thread $i \in \{1, 2\}$

```
1: 'noncritical section'
2: $x := i$
3: if $y = 0$ or $y = i$ then skip else goto 2
4: $z := i$
5: if $x = i$ then skip else goto 2
6: $y := i$
7: if $z = i$ then skip else goto 2
8: 'critical section'
```

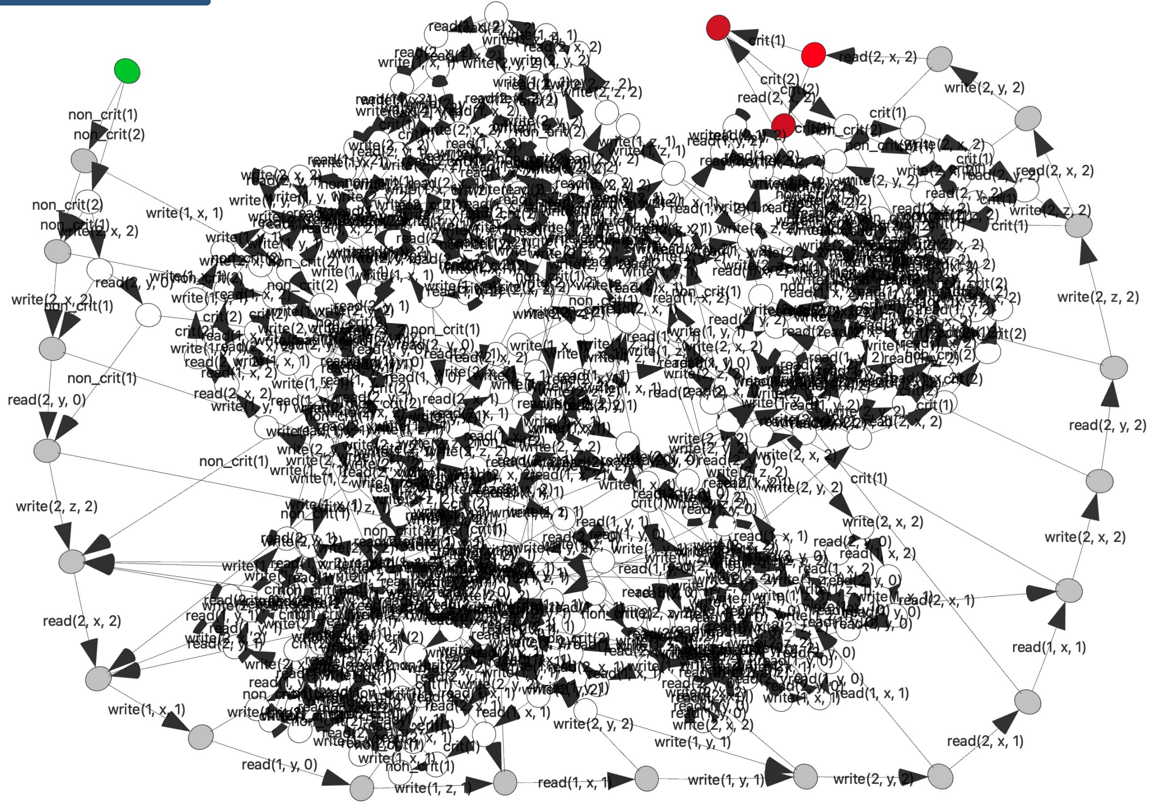


STATE SPACE

0	0	0
x	y	z

The algorithm run by thread $i \in \{1, 2\}$

```
1: 'noncritical section'
2: $x := i$
3: if $y = 0$ or $y = i$ then skip else goto 2
4: $z := i$
5: if $x = i$ then skip else goto 2
6: $y := i$
7: if $z = i$ then skip else goto 2
8: 'critical section'
```



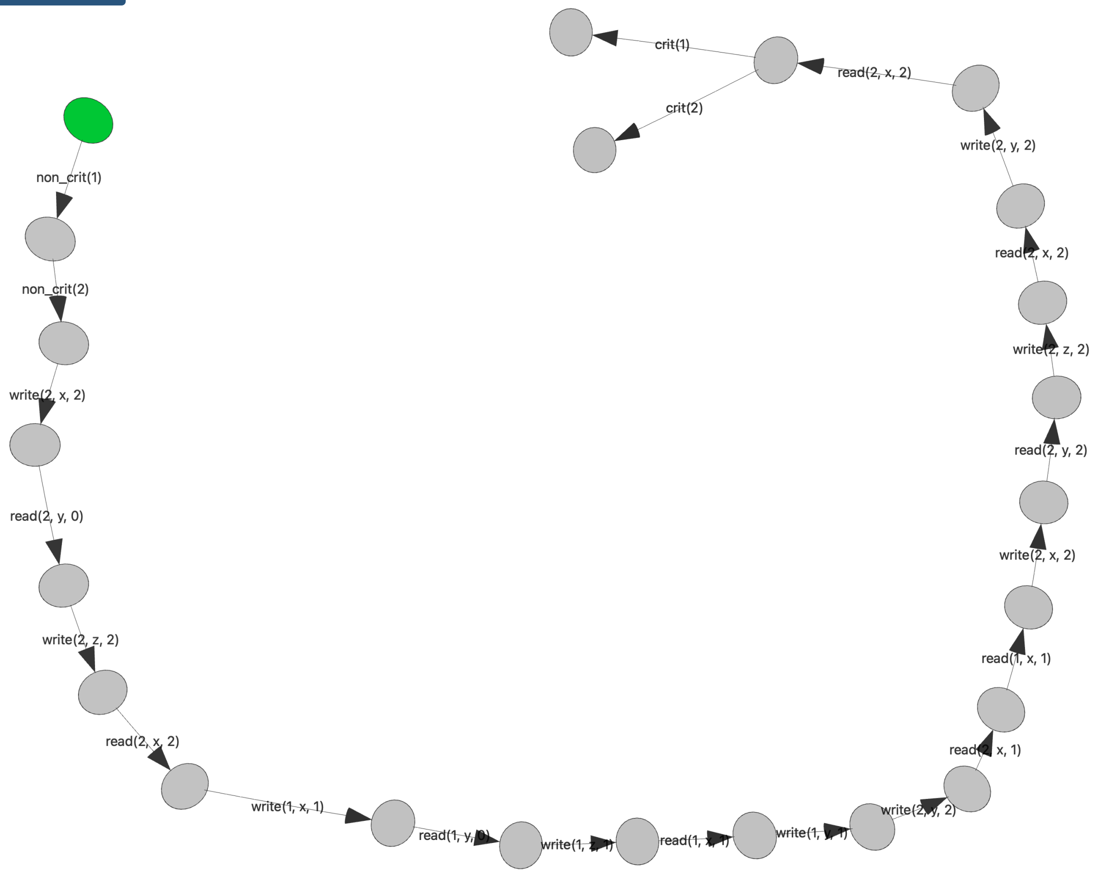
STATE SPACE

0	0	0
x	y	z

The algorithm run by thread $i \in \{1,2\}$

```
1: 'noncritical section'
2: $x := i$
3: if $y = 0$ or $y = i$ then skip else goto 2
4: $z := i$
5: if $x = i$ then skip else goto 2
6: $y := i$
7: if $z = i$ then skip else goto 2
8: 'critical section'
```

mCRL2 analysing system behaviour



COUNTEREXAMPLE

MANY CORRECT MUTUAL EXCLUSION ALGORITHMS

Dekker's mutual exclusion algorithm

Dijkstra's mutual exclusion algorithm

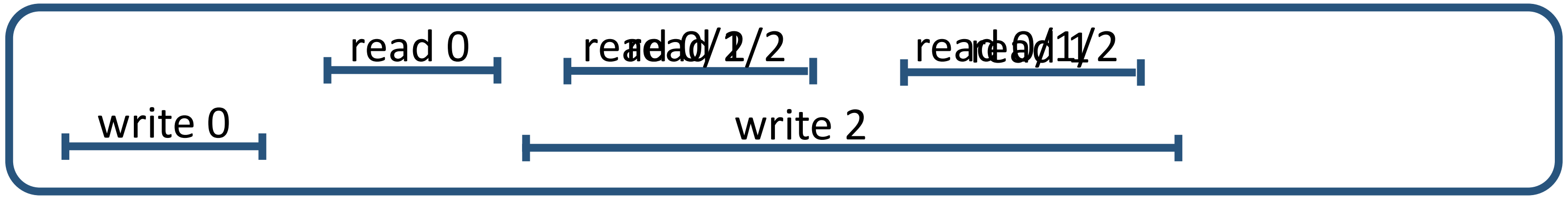
Peterson's mutual exclusion algorithm

Knuth's mutual exclusion algorithm

…

Correctness claims for these algorithms have been established under the assumption that threads interact atomically with shared memory

possible register values: 0,1,2



Leslie Lamport (2013)

Atomicity of memory interaction is not a reasonable assumption for a solution to the mutual exclusion problem

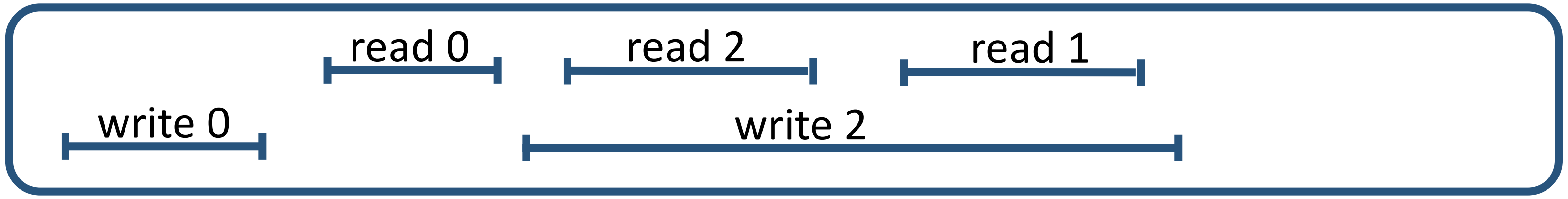
Safe register (a.k.a. *communication variable)*:
    if a read is concurrent with a write, then it may obtain any value in the domain of the register

Bakery Algorithm solves mutual exclusion problem

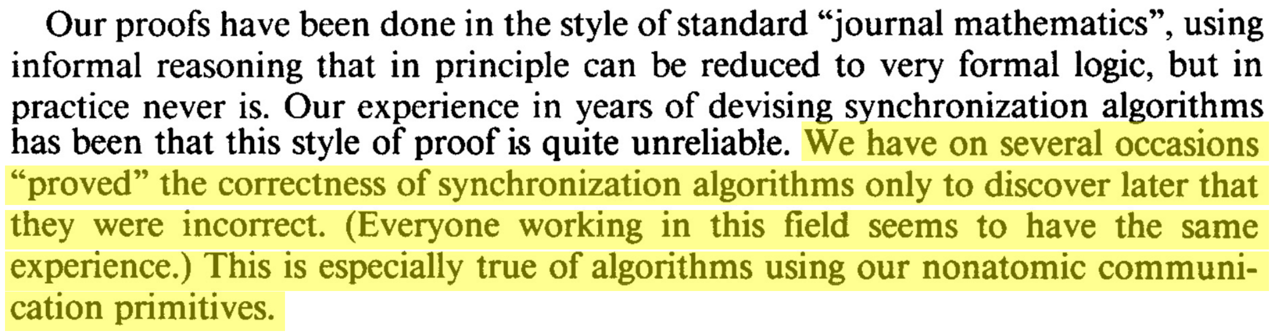
BUT (a.f.a.i.k): this has never been mechanically verified

ATOMICITY?

possible register values: 0,1,2



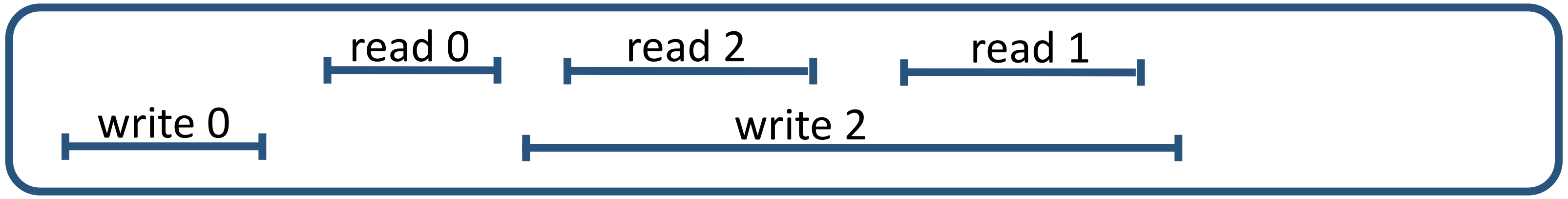
Leslie Lamport (2013)


Our proofs have been done in the style of standard "journal mathematics", using informal reasoning that in principle can be reduced to very formal logic, but in practice never is. Our experience in years of devising synchronization algorithms has been that this style of proof is quite unreliable. We have on several occasions "proved" the correctness of synchronization algorithms only to discover later that they were incorrect. (Everyone working in this field seems to have the same experience.) This is especially true of algorithms using our nonatomic communication primitives.

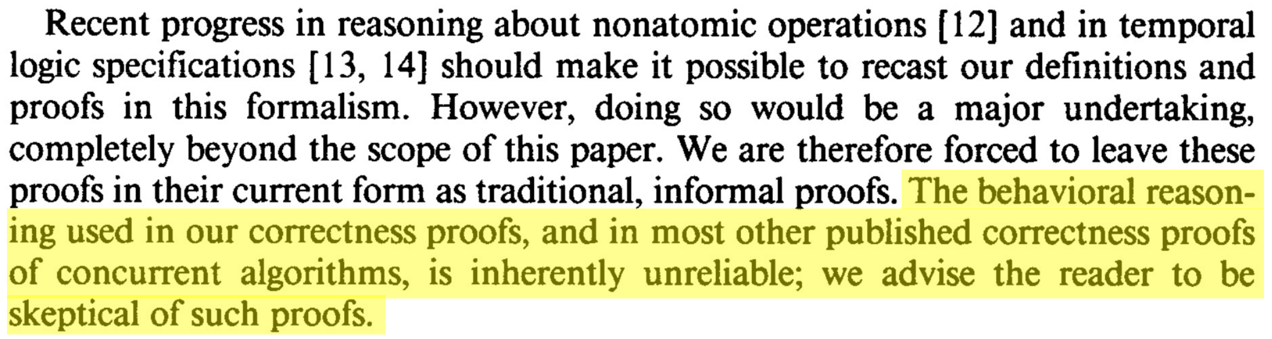
L. Lamport (1986):
The Mutual Exclusion Problem:
Part II---Statement and Solutions
JACM 33(2), pp. 327-348

VERIFY MECHANICALLY!

possible register values: 0,1,2



Leslie Lamport (2013)


Recent progress in reasoning about nonatomic operations [12] and in temporal logic specifications [13, 14] should make it possible to recast our definitions and proofs in this formalism. However, doing so would be a major undertaking, completely beyond the scope of this paper. We are therefore forced to leave these proofs in their current form as traditional, informal proofs. The behavioral reasoning used in our correctness proofs, and in most other published correctness proofs of concurrent algorithms, is inherently unreliable; we advise the reader to be skeptical of such proofs.

L. Lamport (1986):
The Mutual Exclusion Problem:
Part II---Statement and Solutions
JACM 33(2), pp. 327-348

Goal: mechanically verify correctness of mutual exclusion algorithms not relying on atomic communication primitives

VERIFY MECHANICALLY!

Myrthe Spronck (20??)

- Devised general method to model **non-atomic** memory interactions in mCRL2
- Analysed several mutual exclusion protocols claimed to be robust for such interactions
- Lamport only considered **single-writer, multi-reader**; Spronck's definition is suitable for **multi-writer, multi-reader**.

A BACHELOR RESEARCH PROJECT

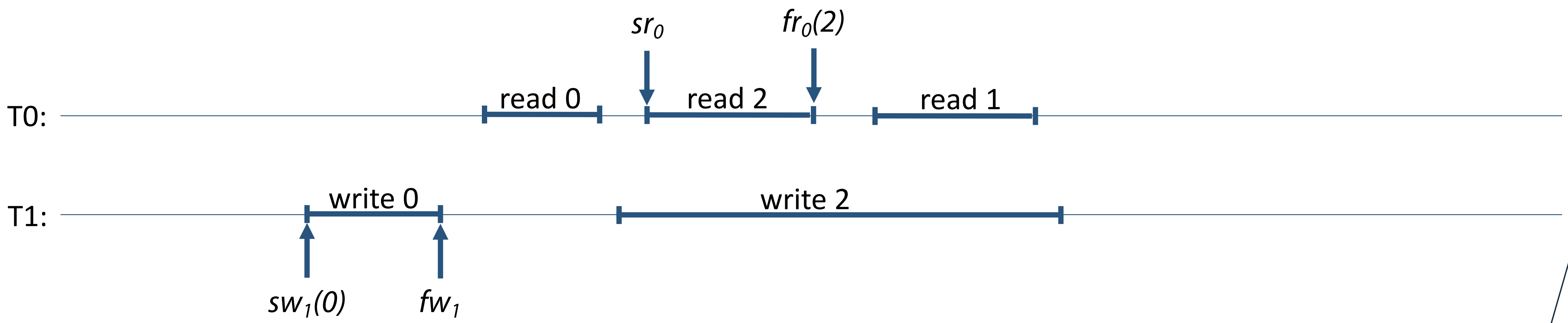
MODELLING REGISTERS WITH NON-ATOMIC OPERATIONS

Main ideas:

- Interactions split up into start and finish
- Register modelled as separate process
- Register keeps track of threads currently reading and writing
- Read overlapping with write:
  ➢ return arbitrary value from domain
- Write overlapping with a write:
  ➢ register assumes arbitrary value from domain

$$R_s(d:\mathbb{D}, s:\mathbb{S}_s)$$

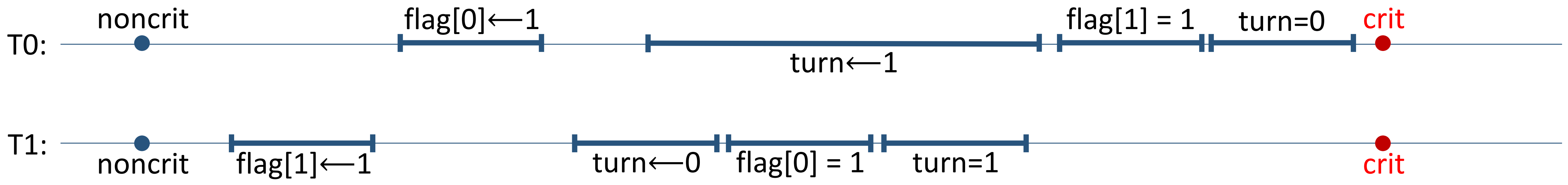
using mCRL2's facility to algebraically specify data types



NB: Peterson **never** claimed that his algorithm is correct also for nonatomic memory interactions!

Counterexample below shown only to illustrate how nonatomic memory interactions influence correctness

```
1: $\mathit{flag}[i] \leftarrow 1$
2: $\mathit{turn} \leftarrow j$
3: await $\mathit{flag}[j] = 0 \lor \mathit{turn} = i$
4: critical section
5: $\mathit{flag}[i] \leftarrow 0$
```



PETERSON (COUNTEREXAMPLE)

Not logged in Talk Contributions Create account Log in

Article Talk Read Edit View history Search Wikipedia

Main page
Contents
Current events
Random article
About Wikipedia
Contact us
Donate

Contribute
Help
Learn to edit
Community portal
Recent changes
Upload file

Tools
What links here
Related changes
Special pages
Permanent link
Page information
Cite this page
Wikidata item

Print/export
Download as PDF
Printable version

Languages
中文

Edit lin

Szymański's algorithm

From Wikipedia, the free encyclopedia

Szymański's Mutual Exclusion Algorithm is a mutual exclusion algorithm devised by computer scientist Dr. Bolesław Szymański, which has many favorable properties including linear wait,[1][2] and which extension[3] solved the open problem posted by Leslie Lamport[4] whether there is an algorithm with a constant number of communication bits per process that satisfies every reasonable fairness and failure-tolerance requirement that Lamport conceived of (Lamport's solution used n factorial communication variables vs. Szymański's 5).

en and the exit door is closed. All processes which request entry into the critical section at roughly the … e waiting room; the last of them closes the entry door and opens the exit door. The processes then

s to leave the critical section closes the exit door and reopens the entry door, so the next batch of

ead by all others (this single-writer property is desirable for efficient cache usage).

w:

e door

ish exit protocol

has
to be closed

a thread other than self. For example, if the test is any flag[1..N] = 1 and only flag[self] = 1, then the test is said to have failed/returned 0. Despite the intuitive explanation, the algorithm was not easy to prove correct,
n presented.[2][5]

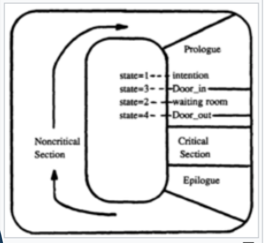
wait". *Proceedings of the 2nd international conference on Supercomputing - ICS '88. ICS '88: Proceedings o* … ISBN 978-0-89791-

Business: A Birthday Salute to Edsger W. Dijkstra. Springer Verlag. pp. 289–301. ISBN 978-0-

n Technology. Jerusalem, Israel: 110–117.

33 (2): 327–348. CiteSeerX 10.1.1.32.9808. doi:10.1145/5383.5385. S2CID 7387839

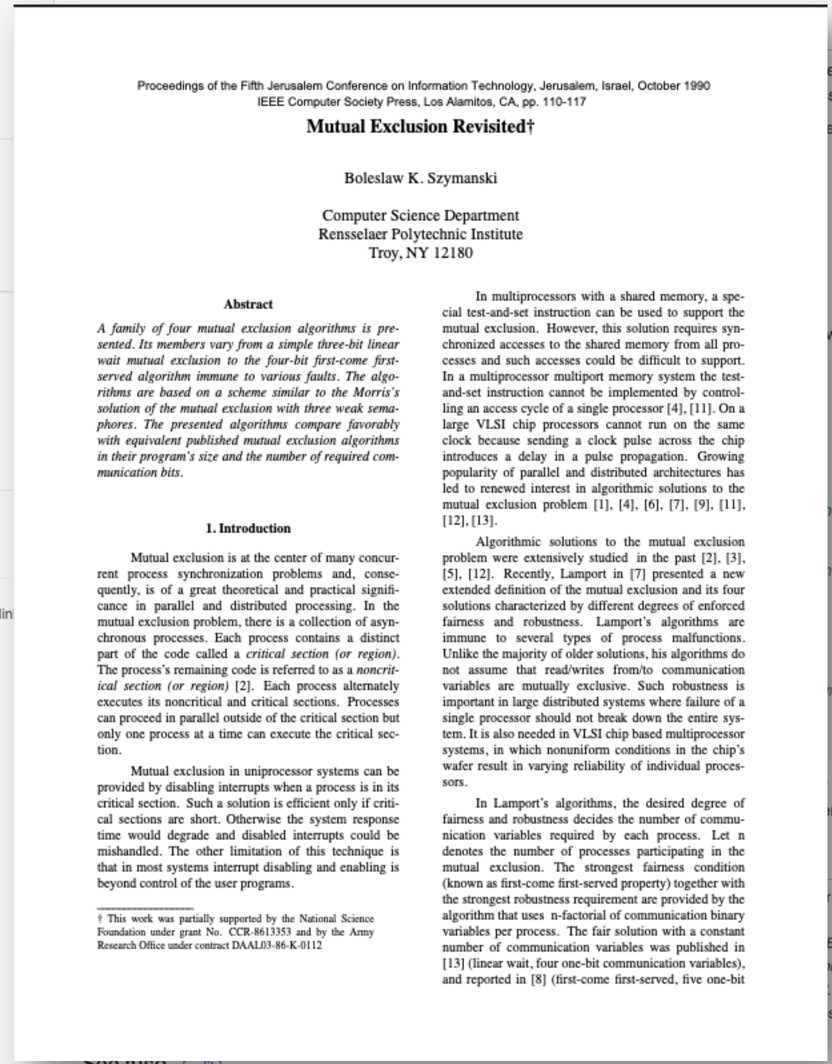
s; Zwiers, Job (November 2001). *Concurrency Verification*. Number 54 in Cambridge Tract



Scheme of Process States During Execution

See also [edit]

- Dekker's algorithm
- Eisenberg & McGuire algorithm
- Peterson's algorithm



Proceedings of the Fifth Jerusalem Conference on Information Technology, Jerusalem, Israel, October 1990
IEEE Computer Society Press, Los Alamitos, CA, pp. 110-117

Mutual Exclusion Revisited†

Boleslaw K. Szymanski

Computer Science Department
Rensselaer Polytechnic Institute
Troy, NY 12180

Abstract

A family of four mutual exclusion algorithms is presented. Its members vary from a simple three-bit linear wait mutual exclusion to the four-bit first-come first-served algorithm immune to various faults. The algorithms are based on a scheme similar to the Morris's solution of the mutual exclusion with three weak semaphores. The presented algorithms compare favorably with equivalent published mutual exclusion algorithms in their program's size and the number of required communication bits.

1. Introduction

Mutual exclusion is at the center of many concurrent process synchronization problems and, consequently, is of a great theoretical and practical significance in parallel and distributed processing. In the mutual exclusion problem, there is a collection of asynchronous processes. Each process contains a distinct part of the code called a *critical section (or region)*. The process's remaining code is referred to as a *noncritical section (or region)* [2]. Each process alternately executes its noncritical and critical sections. Processes can proceed in parallel outside of the critical section but only one process at a time can execute the critical section.

Mutual exclusion in uniprocessor systems can be provided by disabling interrupts when a process is in its critical section. Such a solution is efficient only if critical sections are short. Otherwise the system response time would degrade and disabled interrupts could be mishandled. The other limitation of this technique is that in most systems interrupt disabling and enabling is beyond control of the user programs.

In multiprocessors with a shared memory, a special test-and-set instruction can be used to support the mutual exclusion. However, this solution requires synchronized accesses to the shared memory from all processes and such accesses could be difficult to support. In a multiprocessor multiport memory system the test-and-set instruction cannot be implemented by controlling an access cycle of a single processor [4], [11]. On a large VLSI chip processors cannot run on the same clock because sending a clock pulse across the chip introduces a delay in a pulse propagation. Growing popularity of parallel and distributed architectures has led to renewed interest in algorithmic solutions to the mutual exclusion problem [1], [4], [6], [7], [9], [11], [12], [13].

Algorithmic solutions to the mutual exclusion problem were extensively studied in the past [2], [3], [5], [12]. Recently, Lamport in [7] presented a new extended definition of the mutual exclusion and its four solutions characterized by different degrees of enforced fairness and robustness. Lamport's algorithms are immune to several types of process malfunctions. Unlike the majority of older solutions, his algorithms do not assume that read/writes from/to communication variables are mutually exclusive. Such robustness is important in large distributed systems where failure of a single processor should not break down the entire system. It is also needed in VLSI chip based multiprocessor systems, in which nonuniform conditions in the chip's wafer result in varying reliability of individual processors.

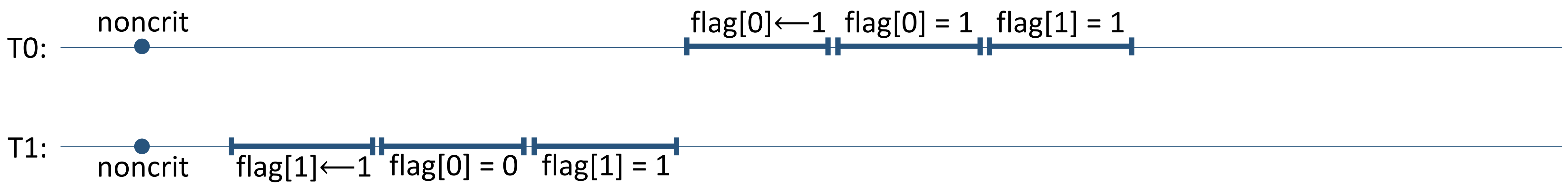
In Lamport's algorithms, the desired degree of fairness and robustness decides the number of communication variables required by each process. Let n denotes the number of processes participating in the mutual exclusion. The strongest fairness condition (known as first-come first-served property) together with the strongest robustness requirement are provided by the algorithm that uses n-factorial of communication binary variables per process. The fair solution with a constant number of communication variables was published in [13] (linear wait, four one-bit communication variables), and reported in [8] (first-come first-served, five one-bit

† This work was partially supported by the National Science Foundation under grant No. CCR-8613353 and by the Army Research Office under contract DAAL03-86-K-0112

"…which extension solved the open problem posted by Leslie Lamport whether there is an algorithm with a constant number of communication bits per process that satisfies every reasonable fairness and failure-tolerance requirement that Lamport conceived of…"

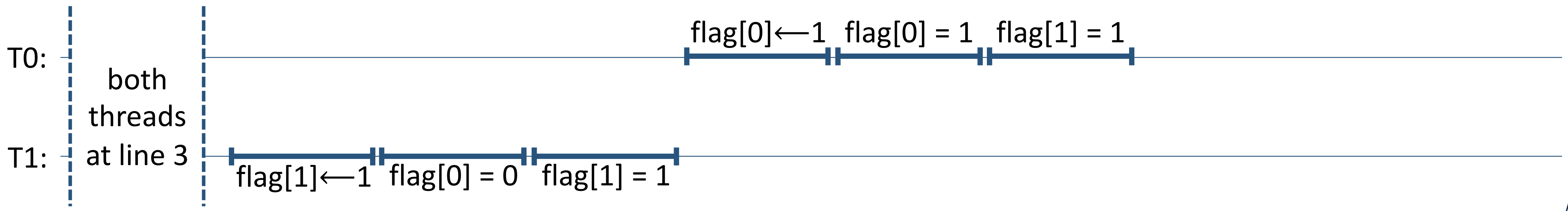
"Despite the intuitive explanation, the algorithm was not easy to prove correct"

```
1:  $flag[i] \leftarrow 1$
2:  await $\forall j.\ flag[j] < 3$
3:  $flag[i] \leftarrow 3$
4:  if $\exists j.\ flag[j] = 1$ then
5:      $flag[i] \leftarrow 2$
6:      await $\exists j.\ flag[j] = 4$
7:  $flag[i] \leftarrow 4$
8:  await $\forall j < i.\ flag[j] < 2$
9:  critical section
10: await $\forall j > i.\ flag[j] < 2 \vee flag[j] > 3$
11: $flag[i] \leftarrow 0$
```

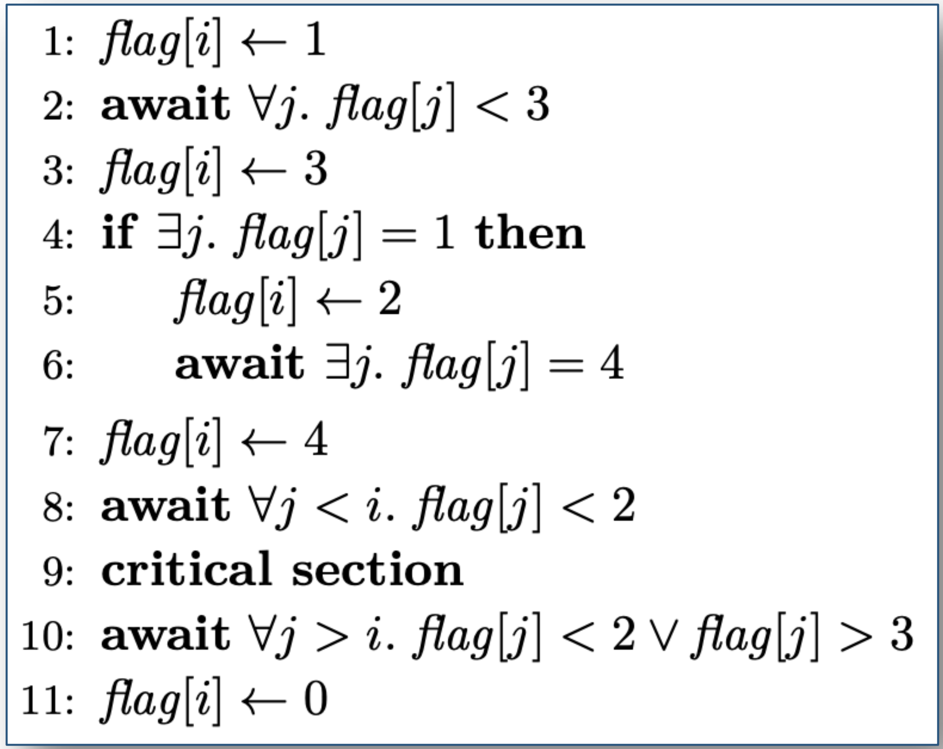


SZYMANSKI (COUNTEREXAMPLE)

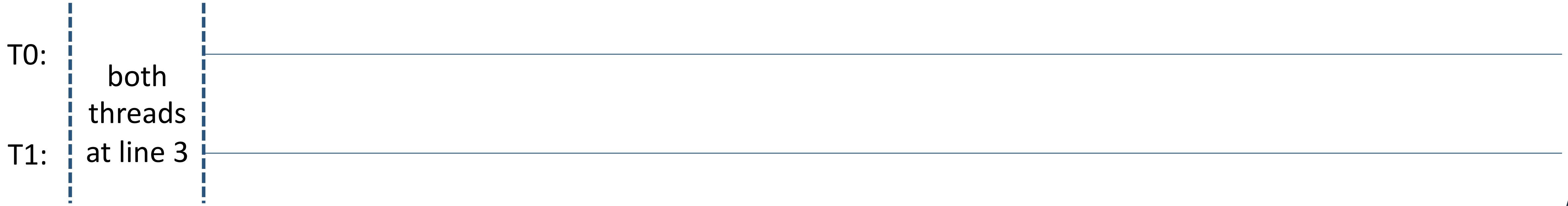
```
1:  $\mathit{flag}[i] \leftarrow 1$
2:  **await** $\forall j.\ \mathit{flag}[j] < 3$
3:  $\mathit{flag}[i] \leftarrow 3$
4:  **if** $\exists j.\ \mathit{flag}[j] = 1$ **then**
5:      $\mathit{flag}[i] \leftarrow 2$
6:      **await** $\exists j.\ \mathit{flag}[j] = 4$
7:  $\mathit{flag}[i] \leftarrow 4$
8:  **await** $\forall j < i.\ \mathit{flag}[j] < 2$
9:  **critical section**
10: **await** $\forall j > i.\ \mathit{flag}[j] < 2 \vee \mathit{flag}[j] > 3$
11: $\mathit{flag}[i] \leftarrow 0$
```



SZYMANSKI (COUNTEREXAMPLE)



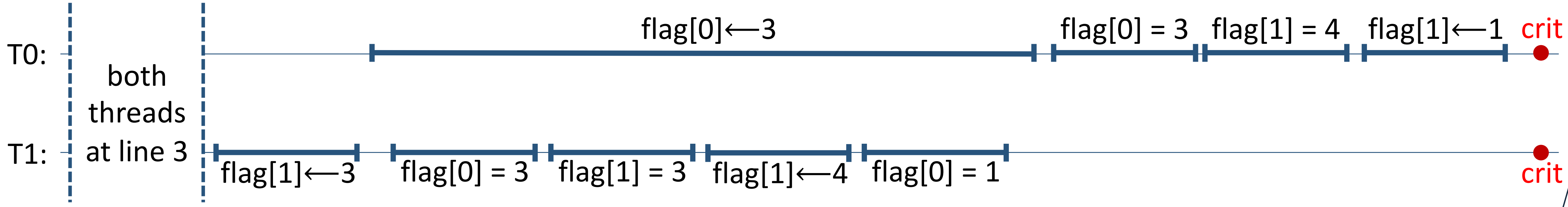
```
1:  $flag[i] \leftarrow 1$
2:  await $\forall j.\ flag[j] < 3$
3:  $flag[i] \leftarrow 3$
4:  if $\exists j.\ flag[j] = 1$ then
5:      $flag[i] \leftarrow 2$
6:      await $\exists j.\ flag[j] = 4$
7:  $flag[i] \leftarrow 4$
8:  await $\forall j < i.\ flag[j] < 2$
9:  critical section
10: await $\forall j > i.\ flag[j] < 2 \vee flag[j] > 3$
11: $flag[i] \leftarrow 0$
```



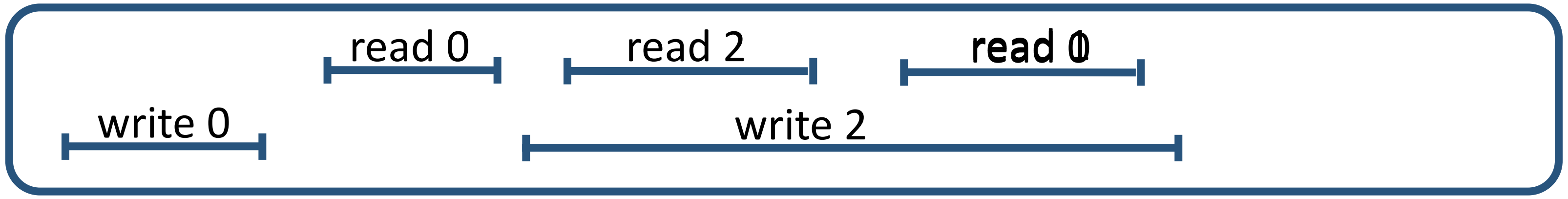
SZYMANSKI (COUNTEREXAMPLE)

SZYMANSKI (COUNTEREXAMPLE)

```
1: $flag[i] \leftarrow 1$
2: await $\forall j.\ flag[j] < 3$
3: $flag[i] \leftarrow 3$
4: if $\exists j.\ flag[j] = 1$ then
5:     $flag[i] \leftarrow 2$
6:     await $\exists j.\ flag[j] = 4$
7: $flag[i] \leftarrow 4$
8: await $\forall j < i.\ flag[j] < 2$
9: critical section
10: await $\forall j > i.\ flag[j] < 2 \lor flag[j] > 3$
11: $flag[i] \leftarrow 0$
```



possible register values: 0,1,2



Leslie Lamport (2013)

Atomicity of memory interaction is not a reasonable assumption for a solution to the mutual exclusion problem

Safe register (a.k.a. *communication variable)*:
if a read is concurrent with a write, then it may obtain any value in the domain of the register

Regular register:
if a read is concurrent with a write, then it may obtain the old or the new value

REGULAR REGISTERS

REGULAR MULTI-WRITER MULTI-READER REGISTER

Main ideas:

- Register modelled as separate process
- Interactions split up into start and finish
- Register keeps track of threads currently reading and writing
- Read overlapping with write:
  ➢ return value written right before or during read
- Write overlapping with a write:
  ➢ non-deterministically fix order of writes at runtime

$$R_r(d : \mathbb{D}, s : \mathbb{S}_r)$$

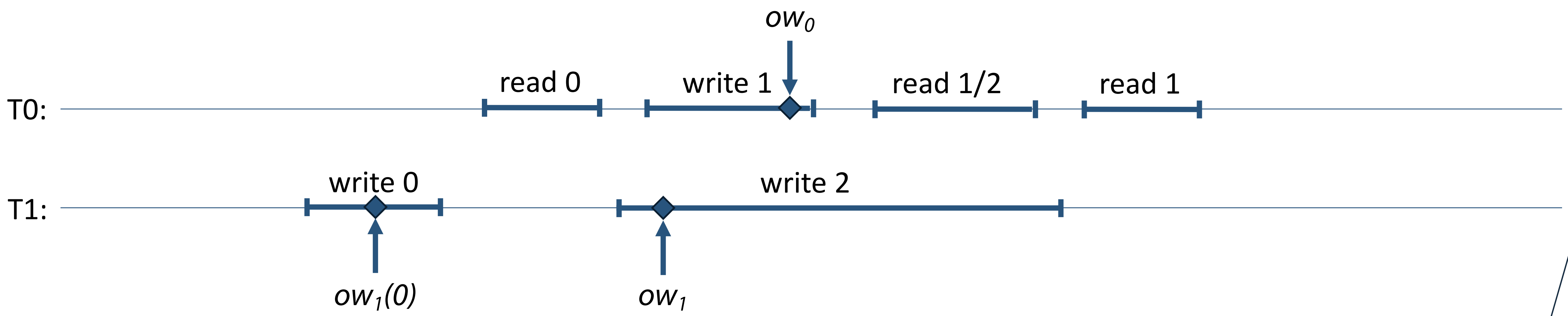
Bookkeeping

for every executing read:

- which values were written concurrently

for every executing write:

- has it been effectuated or not?



REGULAR MULTI-WRITER MULTI-READER REGISTER

Main ideas:

- Register modelled as separate process
- Interactions split up into start and finish
- Register keeps track of threads currently reading and writing
- Read overlapping with write:
  - ➢ return value written right before or during read
- Write overlapping with a write:
  - ➢ non-deterministically fix order of writes at runtime

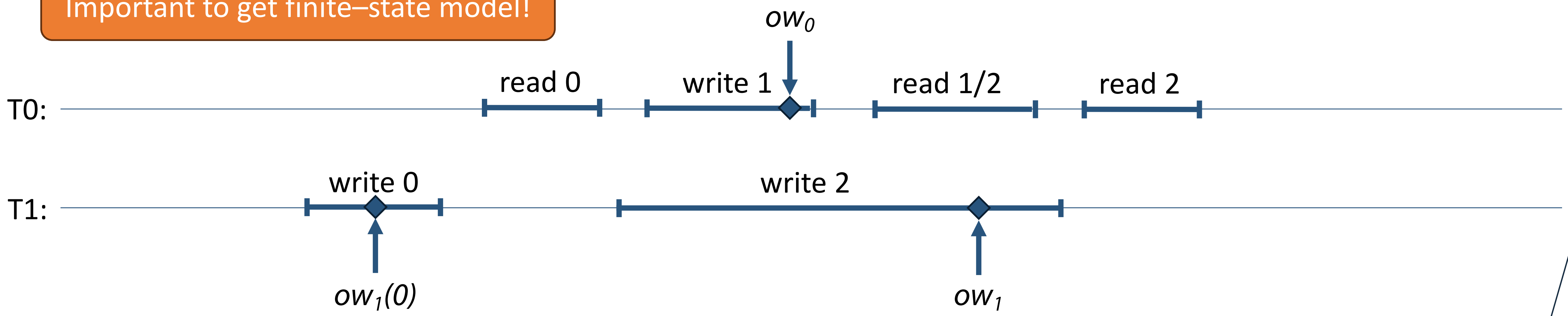
Important to get finite–state model!

$R_r(d : \mathbb{D}, s : \mathbb{S}_r)$

Bookkeeping

for every executing read:
- which values were written concurrently

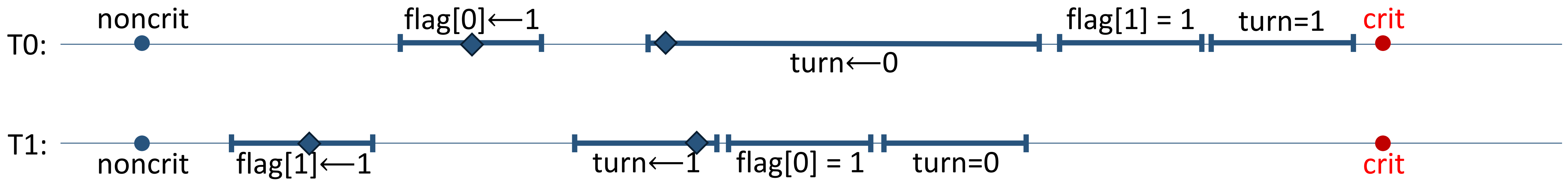
for every executing write:
- has it been effectuated or not?



NB: Peterson **never** claimed that his algorithm is correct also for nonatomic memory interactions!

Counterexample below shown only to illustrate how nonatomic memory interactions influence correctness

```
1: $flag[i] \leftarrow 1$
2: $turn \leftarrow j$
3: **await** $flag[j] = 0 \lor turn = i$
4: **critical section**
5: $flag[i] \leftarrow 0$
```

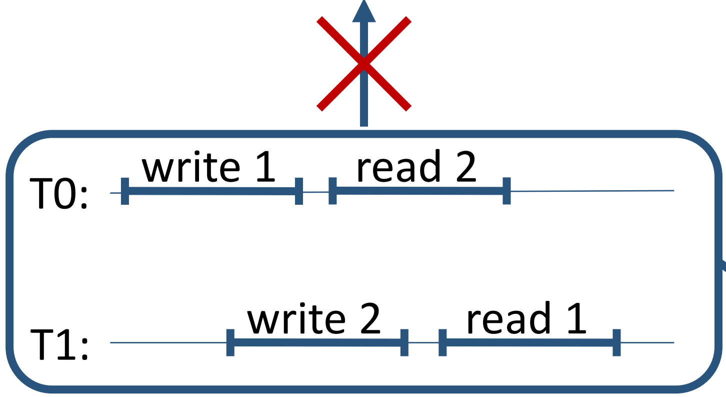


PETERSON (COUNTEREXAMPLE)

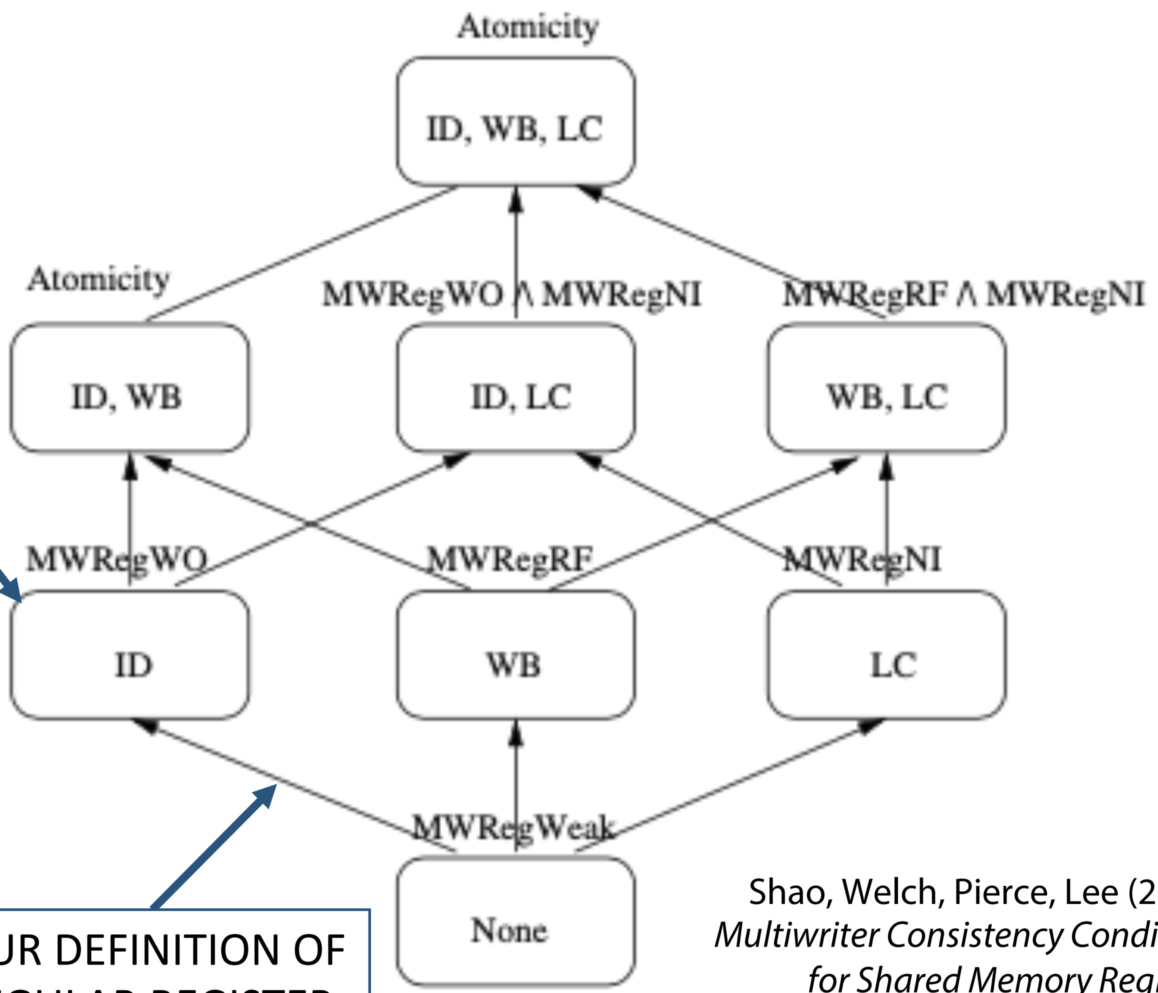
OTHER MWMR REGULAR REGISTER DEFINITIONS

- Definitions given as conditions on computations (cf. fairness assumptions)
- Not directly useful for model checking

A computation satisfies the **write-order** condition if associated with every read r there is a total order S_r on the writes (consistent with the computation) such that S_r and $S_{r'}$ agree on all writes relevant to both



OUR DEFINITION OF REGULAR REGISTER



Shao, Welch, Pierce, Lee (2011): *Multiwriter Consistency Conditions for Shared Memory Registers* SIAM J. Comput. 40(1), pp. 28-62

- Defined generic mCRL2 models for safe, regular and atomic MWMR registers

  see: https://github.com/mCRL2org/mCRL2/tree/master/examples/non-atomic_registers

- Proved relationship with alternative definitions in literature

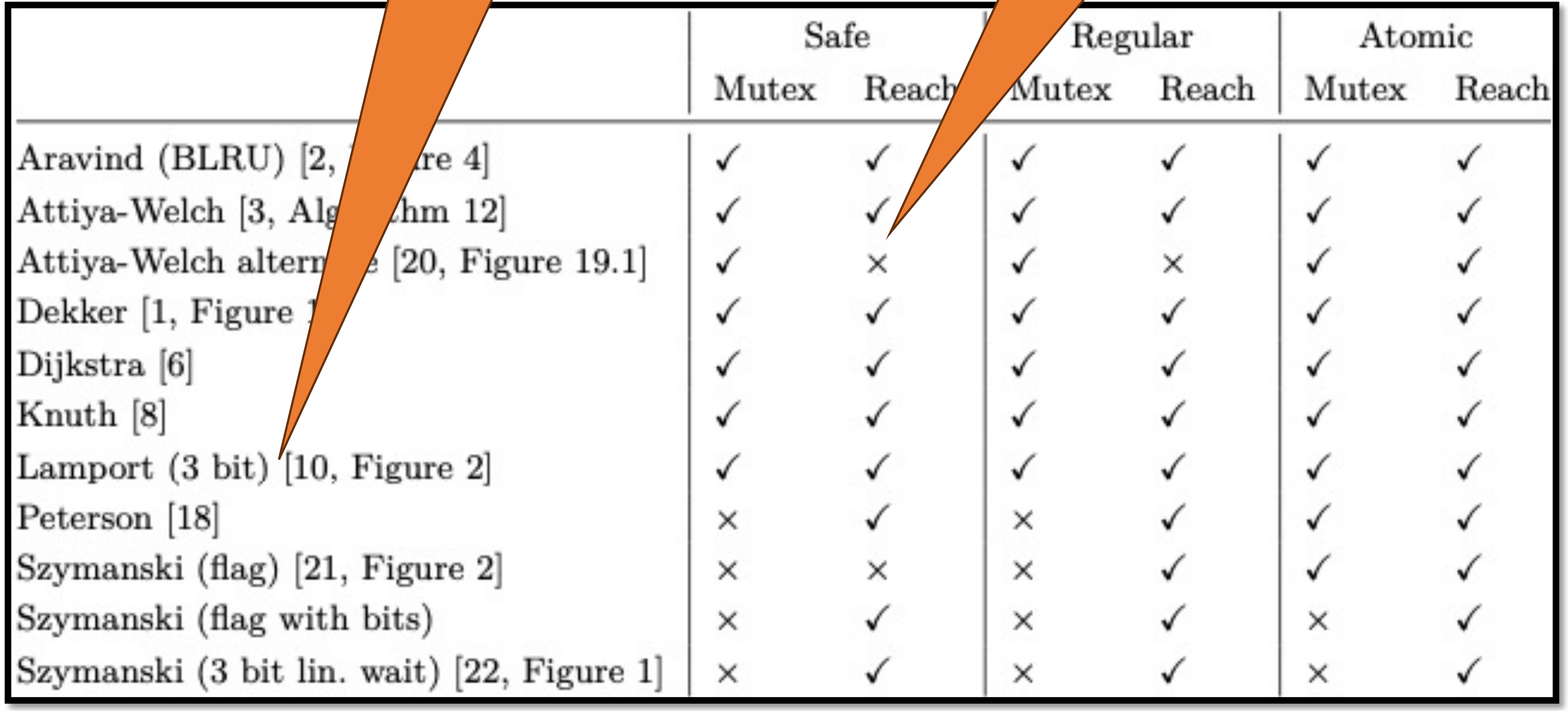
- Verified several well-known mutual exclusion algorithms

- Found issues

CONCLUSION

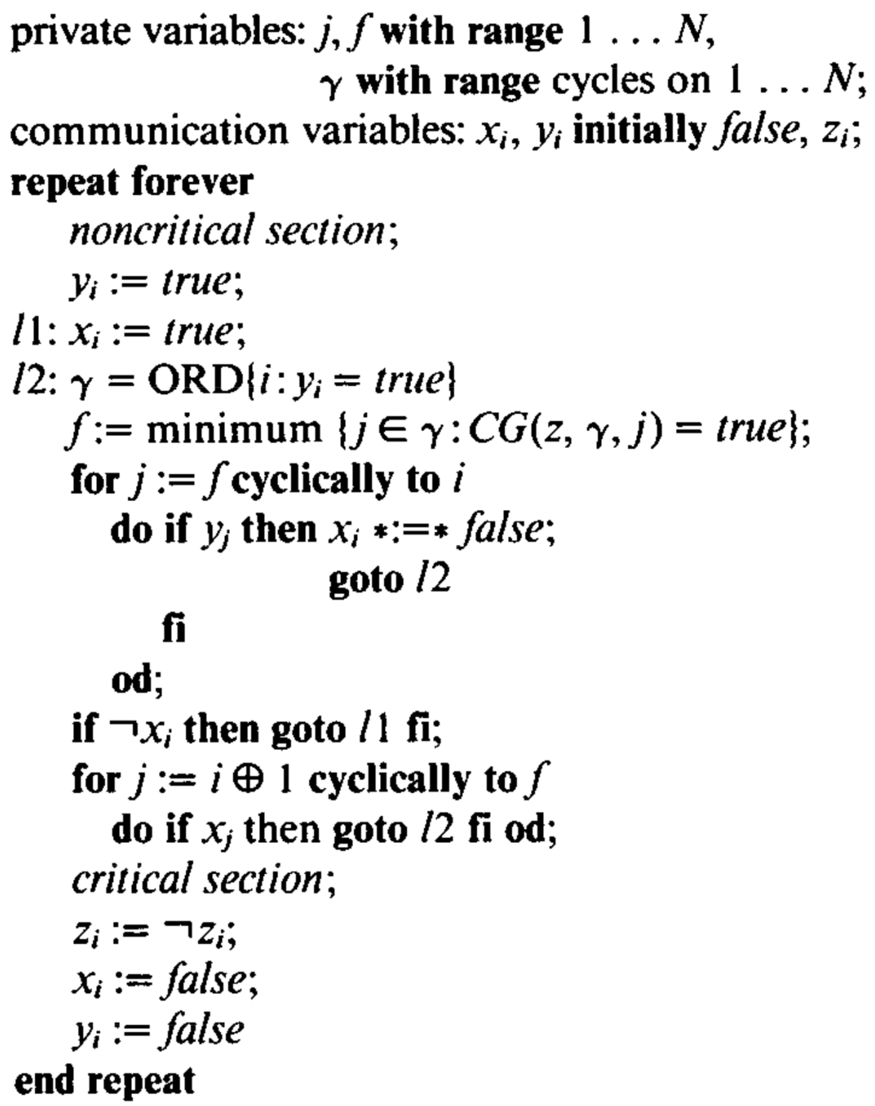
Correctness hinges on subtle implementation detail

Subtle reformulation of the algorithm introduces flaw



	Safe		Regular		Atomic	
	Mutex	Reach	Mutex	Reach	Mutex	Reach
Aravind (BLRU) [2, re 4]	✓	✓	✓	✓	✓	✓
Attiya-Welch [3, Alg hm 12]	✓	✓	✓	✓	✓	✓
Attiya-Welch altern e [20, Figure 19.1]	✓	×	✓	×	✓	✓
Dekker [1, Figure 1	✓	✓	✓	✓	✓	✓
Dijkstra [6]	✓	✓	✓	✓	✓	✓
Knuth [8]	✓	✓	✓	✓	✓	✓
Lamport (3 bit) [10, Figure 2]	✓	✓	✓	✓	✓	✓
Peterson [18]	×	✓	×	✓	✓	✓
Szymanski (flag) [21, Figure 2]	×	×	×	✓	✓	✓
Szymanski (flag with bits)	×	✓	×	✓	×	✓
Szymanski (3 bit lin. wait) [22, Figure 1]	×	✓	×	✓	×	✓

CONCLUSIONS



```
private variables: $j, f$ with range $1 \ldots N$,
                   $\gamma$ with range cycles on $1 \ldots N$;
communication variables: $x_i$, $y_i$ initially $false$, $z_i$;
repeat forever
   noncritical section;
   $y_i := true$;
$l1$: $x_i := true$;
$l2$: $\gamma = \mathrm{ORD}\{i : y_i = true\}$
   $f :=$ minimum $\{j \in \gamma : CG(z, \gamma, j) = true\}$;
   for $j := f$ cyclically to $i$
      do if $y_j$ then $x_i$ $*{:=}*$ $false$;
                  goto $l2$
         fi
      od;
   if $\neg x_i$ then goto $l1$ fi;
   for $j := i \oplus 1$ cyclically to $f$
      do if $x_j$ then goto $l2$ fi od;
   critical section;
   $z_i := \neg z_i$;
   $x_i := false$;
   $y_i := false$
end repeat
```

LAMPORT'S THREE BIT ALGORITHM

ATTIYA-WELCH

```
1: $flag[i] \leftarrow 0$
2: await $flag[j] = 0 \vee turn = j$
3: $flag[i] \leftarrow 1$
4: if $turn = i$ then
5:     if $flag[j] = 1$ then
6:         goto line 1
7: else
8:     await $flag[j] = 0$
9: critical section
10: $turn \leftarrow i$
11: $flag[i] \leftarrow 0$
```

```
1: repeat
2:     $flag[i] \leftarrow 0$
3:     await $flag[j] = 0 \vee turn = j$
4:     $flag[i] \leftarrow 1$
5: until $turn = j \vee flag[j] = 0$
6: if $turn = j$ then
7:     await $flag[j] = 0$
8: critical section
9: $turn \leftarrow i$
10: $flag[i] \leftarrow 0$
```

1. Are the regular register models by Shao et al. finite-state?

2. If not, can we incorporate their conditions in modal mu-calculus formulas?

3. Is Peterson's algorithm correct with respect to the regular registers of Shao et al.?

4. How to formulate fairness assumptions to verify starvation freedom?

5. Model other types of failures

6. …

OPEN PROBLEMS