The Bakery Algorithm in 2015

Leslie Lamport
Microsoft Research

The Algorithm in 1974

The Algorithm in 1974

A mutual exclusion algorithm for processes named 1, . . . , N.

The Algorithm in 1974

A mutual exclusion algorithm for processes named 1, . . . , N.

Here's the algorithm for process i . . .

The Algorithm in 1974

```
L1: choosing[i] := 1;
    number[i] := 1 + maximum(number[1], . . . , number[N]);
    choosing[i] := 0;
    for j = 1 step l until N do
      begin
        L2: if choosing[j] /= 0 then goto L2;
        L3: if number[j] /= 0 and
                (number[j], j) < (number[i], i)
            then goto L3;
      end;
    critical section;
    number[i] := O;
    noncritical section;
    goto L1;
```

The Algorithm in 1974

Don't try to read it now.

```
L1: choosing[i] := 1;
    number[i] := 1 + maximum(number[1], . . . , number[N]);
    choosing[i] := 0;
    for j = 1 step l until N do
      begin
        L2: if choosing[j] /= 0 then goto L2;
        L3: if number[j] /= 0 and
                (number[j], j) < (number[i], i)
            then goto L3;
      end;
    critical section;
    number[i] := O;
    noncritical section;
    goto L1;
```

The Algorithm in 1974

This pseudocode is old-fashioned, so I'll update it.

```
L1: choosing[i] := 1;
    number[i] := 1 + maximum(number[1], . . . , number[N]);
    choosing[i] := 0;
    for j = 1 step l until N do
      begin
        L2: if choosing[j] /= 0 then goto L2;
        L3: if number[j] /= 0 and
                (number[j], j) < (number[i], i)
            then goto L3;
      end;
    critical section;
    number[i] := O;
    noncritical section;
    goto L1;
```

The Algorithm in 1974

For example, **goto**s were old-fashioned in 1974.

```
L1: choosing[i] := 1;
    number[i] := 1 + maximum(number[1], ..., number[N]);
    choosing[i] := 0;
    for j = 1 step 1 until N do
      begin
        L2: if choosing[j] /= 0 then goto L2;
        L3: if number[j] /= 0 and
                (number[j], j) < (number[i], i)
            then goto L3;
      end;
    critical section;
    number[i] := 0;
    noncritical section;
    goto L1;
```

The Algorithm in 1974

```
while true do
  begin
    choosing[i] := 1;
    number[i] := 1 + maximum(number[1], ..., number[N]);
    choosing[i] := 0;
    for j = 1 step 1 until N do
      begin
        L2: if choosing[j] /= 0 then goto L2;
        L3: if number[j] /= 0 and
                (number[j], j) < (number[i], i)
            then goto L3;
      end;
    critical section;
    number[i] := O;
    noncritical section;
  end
```

The Algorithm in 1974

A number of such changes yield. . .

```
while true do
  begin
    choosing[i] := 1;
    number[i] := 1 + maximum(number[1], . . . , number[N]);
    choosing[i] := 0;
    for j = 1 step 1 until N do
      begin
        L2: if choosing[j] /= 0 then goto L2;
        L3: if number[j] /= 0 and
                (number[j], j) < (number[i], i)
            then goto L3;
      end;
    critical section;
    number[i] := O;
    noncritical section;
  end
```

The Algorithm I Might Have Written in 1979

```
while true do
  begin
    noncritical section;
    flag[i] := true;
    num[i] := any nat > maximum(num[1], ..., num[N]);
    flag[i] := false;
    for all j in {1, ..., N} except j=i do
      begin
        wait for not flag[j];
        wait for num[j] = 0 or
                  (num[i], i) < (num[j], j);
      end;
    critical section;
    num[i] := O;
  end
```

The Algorithm I Might Have Written in 1979

```
while true do
  begin
    noncritical section;
    flag[i] := true;
    num[i] := any nat > maximum(num[1], ..., num[N]);
    flag[i] := false;
    for all j in {1, ..., N} except j=i do
      begin
        wait for not flag[j];
        wait for num[j] = 0 or
                  (num[i], i) < (num[j], j);
      end;
    critical section;
    num[i] := 0;
  end
```

None of these changes needed for what follows.

In the 1980s

In the 1980s

People realized that the only reliable general method for reasoning about concurrent algorithms was with an inductive invariant.

In the 1980s

People realized that the only reliable general method for reasoning about concurrent algorithms was with an inductive invariant.

There are tricks that work for particular algorithms, including the bakery algorithm.

In the 1980s

People realized that the only reliable general method for reasoning about concurrent algorithms was with an inductive invariant.

A state function that cannot be made false by **any** program step.

In the 1980s

People realized that the only reliable general method for reasoning about concurrent algorithms was with an inductive invariant.

An inductive invariant must refer to the control state.

In the 1980s

People realized that the only reliable general method for reasoning about concurrent algorithms was with an inductive invariant.

An inductive invariant must refer to the control state.

Adding labels makes it easy to describe the control state.

The Algorithm in 1979

```
while true do
  begin
    noncritical section;
    flag[i] := true;
    num[i] := any nat > maximum(num[1], ..., num[N]);
    flag[i] := false;
    for all j in {1, ..., N} except j=i do
      begin
        wait for not flag[j];
        wait for num[j] = 0 or
                  (num[i], i) < (num[j], j);
      end;
    critical section;
    num[i] := 0;
  end
```

The Algorithm in 1979

Adding labels yields. . .

```
while true do
  begin
    noncritical section;
    flag[i] := true;
    num[i] := any nat > maximum(num[1], ..., num[N]);
    flag[i] := false;
    for all j in {1, ..., N} except j=i do
      begin
        wait for not flag[j];
        wait for num[j] = 0 or
                  (num[i], i) < (num[j], j);
      end;
    critical section;
    num[i] := 0;
  end
```

The Algorithm in the 1980s

```
while true do
  begin
    ncs: noncritical section;
     e1: flag[i] := true;
     e3: num[i] :=
          e2: any nat > maximum(num[1], ..., num[N]);
     e4: flag[i] := false;
         for all j in {1, ..., N} except j=i do
           begin
         w1: wait for not flag[j];
         w2: wait for num[j] = 0 or
                        (num[i], i) < (num[j], j);
           end;
     cs: critical section;
   exit: num[i] := 0;
  end
```

The Algorithm in 2015

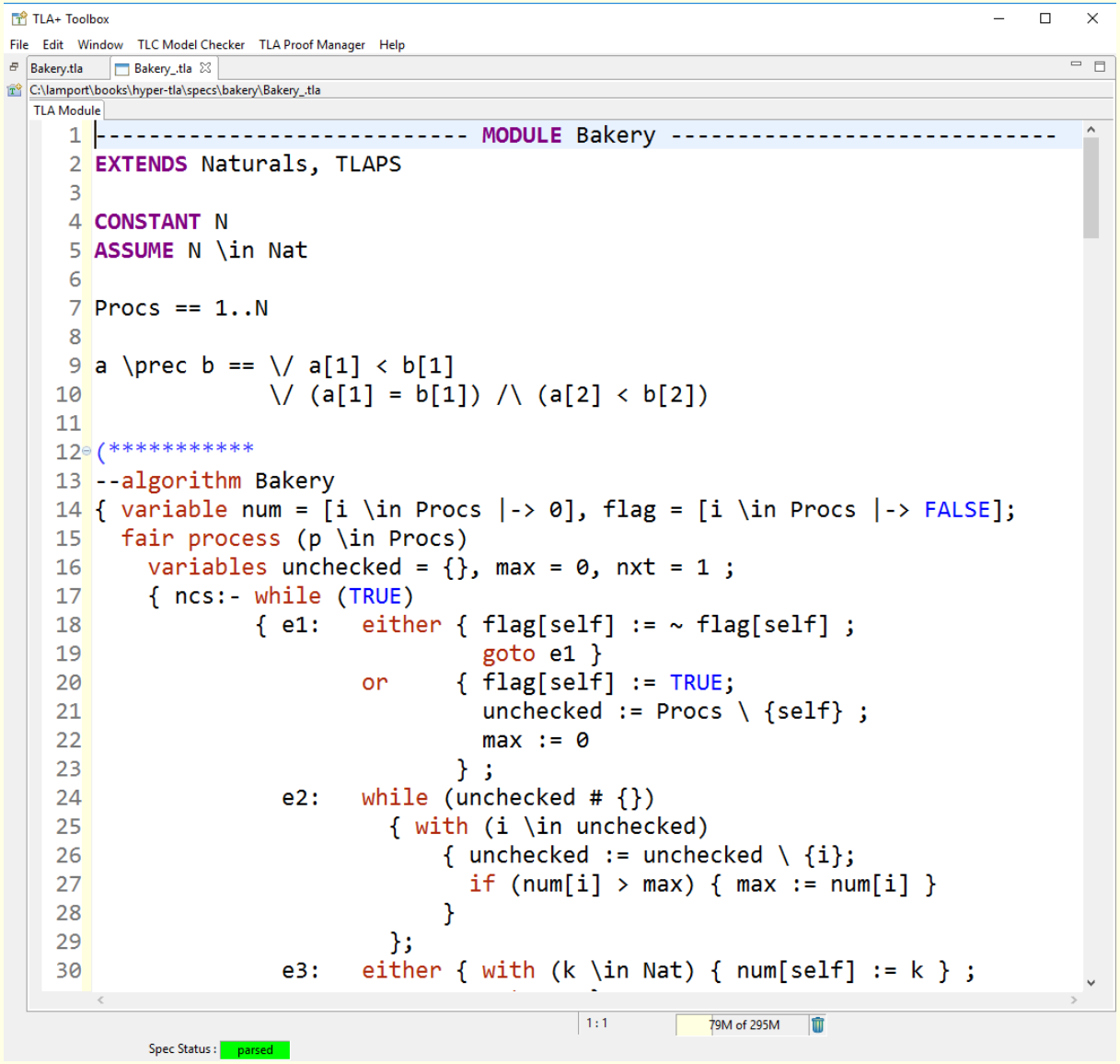
In 2015 I wrote a PlusCal specification and a correctness proof of the algorithm.

The Algorithm in 2015

In 2015 I wrote a PlusCal specification and a correctness proof of the algorithm.

I'll explain why later.

I typed it in ASCII in the TLA+ Toolbox



Automatically Pretty Printed

```
MODULE Bakery
EXTENDS Naturals, TLAPS

CONSTANT N
ASSUME N ∈ Nat

Procs ≜ 1 .. N

a ≺ b ≜  ∨ a[1] < b[1]
         ∨ (a[1] = b[1]) ∧ (a[2] < b[2])

(**********
--algorithm Bakery
{ variable num = [i ∈ Procs ↦ 0], flag = [i ∈ Procs ↦ FALSE] ;
  process ( p ∈ Procs )
    variables unchecked = {}, max = 0, nxt = 1 ;
    { ncs: while ( TRUE )
             { e1:  either { flag[self] := ¬flag[self] ;
                             goto e1 }
                    or     { flag[self] := TRUE ;
                             unchecked := Procs \ {self} ;
                             max := 0
                           } ;
               e2:  while ( unchecked ≠ {} )
                      { with ( i ∈ unchecked )
                          { unchecked := unchecked \ {i} ;
                            if ( num[i] > max ) { max := num[i] }
                          }
                      } ;
               e3:  either { with ( k ∈ Nat ) { num[self] := k } } ;
```

Automatically Pretty Printed

I'll show the pretty-printed version.

The Algorithm In PlusCal

The Algorithm In PlusCal

MODULE *Bakery*

The PlusCal algorithm appears in a TLA$^+$ module.

The Algorithm In PlusCal

MODULE *Bakery*

EXTENDS *Naturals*, *TLAPS*

Imports standard modules.

The Algorithm In PlusCal

MODULE *Bakery*

EXTENDS *Naturals*, *TLAPS*

CONSTANT N
ASSUME $N \in Nat$

Declares N (the number of processes) and what we assume about it.

The Algorithm In PlusCal

MODULE *Bakery*

EXTENDS *Naturals*, *TLAPS*

CONSTANT N
ASSUME $N \in Nat$

$Procs \triangleq 1 \,..\, N$

Defines $Procs$ to be the set of process names.

The Algorithm In PlusCal

MODULE *Bakery*

EXTENDS *Naturals*, *TLAPS*

CONSTANT N
ASSUME $N \in Nat$

$Procs \triangleq 1 \,..\, N$

In 1979, I used $<$ to mean lexicographical ordering of pairs.

The Algorithm In PlusCal

MODULE *Bakery*

EXTENDS *Naturals*, *TLAPS*

CONSTANT N
ASSUME $N \in Nat$

$Procs \triangleq 1 \,..\, N$

In PlusCal / TLA$^+$, $<$ means inequality of numbers.

The Algorithm In PlusCal

MODULE *Bakery*

EXTENDS *Naturals*, *TLAPS*

CONSTANT N
ASSUME $N \in Nat$

$Procs \triangleq 1 \,..\, N$

$$a \prec b \triangleq \begin{array}{l} a[1] < b[1] \\ \lor\ (a[1] = b[1]) \land (a[2] < b[2]) \end{array}$$

Defines $\prec$ to be mean lexicographical ordering of pairs.

(In PlusCal / TLA$^+$, an ordered pair p has components $p[1]$ and $p[2]$.)

```
(**********
```

The algorithm appears inside a comment.

(**********

--algorithm $Bakery$

I called the algorithmn $Bakery$.

```
(***********
--algorithm Bakery
{ variables
```

Declared the global variables.

```
(***********
--algorithm Bakery
{ variables num
```

Declaration of num

$(***********$

--algorithm $Bakery$

{ variables $num = [i \in Procs \mapsto 0]$

Declaration of num and its initial value.

```
(**********
--algorithm Bakery
{ variables num = [i ∈ Procs ↦ 0]
```

The array A with index set $Procs$ and $A[i] = 0$ for all $i \in Procs$.

```
(***********
--algorithm Bakery
{ variables num = [i ∈ Procs ↦ 0]
```

function domain

The ~~array~~ A with ~~index set~~ $Procs$

and $A[i] = 0$ for all $i \in Procs$.

$(***********$

--algorithm $Bakery$

{ variables $num = [i \in Procs \mapsto 0]$, $flag = [i \in Procs \mapsto \text{FALSE}]$ **;**

The declaration and initial value of $flag$.

```
(***********
--algorithm Bakery
{ variables num = [i ∈ Procs ↦ 0], flag = [i ∈ Procs ↦ FALSE] ;
  process ( p ∈ Procs )
```

Begins the specification of a set of processes, with one process for every element of $Procs$.

```
(***********
--algorithm Bakery
{ variables num = [i ∈ Procs ↦ 0], flag = [i ∈ Procs ↦ FALSE] ;
  process ( p ∈ Procs )
    variables
```

Declares the variables local to each process.

```
(***********
--algorithm Bakery
{ variables num = [i ∈ Procs ↦ 0], flag = [i ∈ Procs ↦ FALSE] ;
  process ( p ∈ Procs )
    variables unchecked = {}
```

Variable $unchecked$ initially equal to the empty set

```
(***********
--algorithm Bakery
{ variables num = [i ∈ Procs ↦ 0], flag = [i ∈ Procs ↦ FALSE] ;
  process ( p ∈ Procs )
    variables unchecked = {}, max = 0
```

Variable $unchecked$ initially equal to the empty set

and max

```
(***********
--algorithm Bakery
{ variables num = [i ∈ Procs ↦ 0], flag = [i ∈ Procs ↦ FALSE] ;
  process ( p ∈ Procs )
    variables unchecked = {}, max = 0, nxt = 1 ;
```

Variable $unchecked$ initially equal to the empty set and max and nxt .

```
(***********
--algorithm Bakery
{ variables num = [i ∈ Procs ↦ 0], flag = [i ∈ Procs ↦ FALSE] ;
  process ( p ∈ Procs )
    variables unchecked = {}, max = 0, nxt = 1 ;
```

Next comes the body of the processes.

This part of the pseudocode

```
while true do
  begin
```

This part of the pseudocode

```
while true do
  begin
```

represented by

```
{       while ( TRUE )
         {
```

This part of the pseudocode

```
while true do
  begin
    ncs: noncritical section;
     e1:
```

represented by

```
{       while ( TRUE )
         {
```

This part of the pseudocode

```
while true do
  begin
    ncs: noncritical section;
     e1:
```

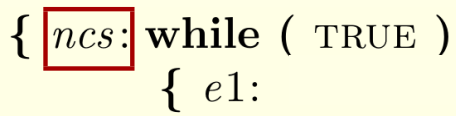
represented by

$$\{\ ncs\colon \textbf{while}\ (\ \textsc{true}\ )\ \{\ e1\colon$$

This part of the pseudocode

```
while true do
  begin
    ncs: noncritical section;
     e1:
```

represented by



In PlusCal, an atomic action is execution from one label

This part of the pseudocode

```
while true do
  begin
    ncs: noncritical section;
     e1:
```

represented by

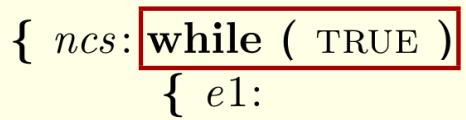
$$\{\ ncs\colon \textbf{while}\ (\ \text{TRUE}\ )\ \{\ e1\colon$$

In PlusCal, an atomic action is execution from one label to the next.

This part of the pseudocode

```
while true do
  begin
    ncs: noncritical section;
     e1:
```

represented by



In PlusCal, an atomic action is execution from one label to the next.

The noncritical section is represented by the **while** test.

This part of the pseudocode

```
e1: flag[i] := true;
e3:
```

This part of the pseudocode

```
e1: flag[i] := true;
e3:
```

The bakery algorithm assumes only safe registers.

This part of the pseudocode

```
e1: flag[i] := true;
e3:
```

The bakery algorithm assumes only safe registers.

A safe register allows a read that overlaps a write to obtain any legal value.

This part of the pseudocode

```
e1: flag[i] := true;
e3:
```

We model a safe register r as follows:

This part of the pseudocode

```
e1: flag[i] := true;
e3:
```

We model a safe register r as follows:

To write v to r :

This part of the pseudocode

```
e1: flag[i] := true;
e3:
```

We model a safe register r as follows:

To write v to r:

– Perform a sequence of atomic writes of arbitrary legal values to r.

This part of the pseudocode

```
e1: flag[i] := true;
e3:
```

We model a safe register r as follows:

To write v to r:

- Perform a sequence of atomic writes of arbitrary legal values to r.
- Atomically write v to r.

This part of the pseudocode

```
e1: flag[i] := true;
e3:
```

We model a safe register r as follows:

- To write v to r:
  - Perform a sequence of atomic writes of arbitrary legal values to r.
  - Atomically write v to r.
- Read r atomically.

This part of the pseudocode

```
e1: flag[i] := true;
e3:
```

We model a safe register r as follows:

- To write v to r:
  - Perform a sequence of atomic writes of arbitrary legal values to r.
  - Atomically write v to r.
- Read r atomically.

This model captures the semantics of a safe register.

This part of the pseudocode

```
e1: flag[i] := true;
e3:
```

This part of the pseudocode

```
e1: flag[i] := true;
e3:
```

represented by

$e1$:

$e2$:

This part of the pseudocode

```
e1: flag[i] := true;
e3:
```

represented by

$e1$:

$e2$:

Yes, we'll see soon that this is right.

This part of the pseudocode

```
e1: flag[i] := true;
e3:
```

represented by

```
e1:  either {
                    }
     or     {


          } ;
e2:
```

Nondeterministically choose which clause to execute.

This part of the pseudocode

```
e1: flag[i] := true;
e3:
```

represented by

$e1$: **either** { $flag[self] := \neg flag[self]$; }
or {

} ;
$e2$:

$self$ is the name of the current process.

This part of the pseudocode

```
e1: flag[i] := true;
e3:
```

represented by

$e1$: **either** { $flag[self] := \neg flag[self]$; }
or {

} ;

$e2$:

Complement $flag$

This part of the pseudocode

```
e1: flag[i] := true;
e3:
```

represented by

$e1$: **either** { $flag[self] := \neg flag[self]$;
goto $e1$ }
or {

} ;
$e2$:

Complement $flag$ and repeat.

This part of the pseudocode

```
e1: flag[i] := true;
e3:
```

represented by

$e1$: **either** { $flag[self] := \neg flag[self]$;
goto $e1$ }
or {

} ;
$e2$:

Complement $flag$ and repeat.

(The only legal values of $flag[self]$ are TRUE and FALSE.)

This part of the pseudocode

```
e1: flag[i] := true;
e3:
```

represented by

$e1$: **either** { $flag[self] := \neg flag[self]$;
goto $e1$ }
or { $flag[self] :=$ TRUE ;

} ;
$e2$:

or set $flag[self]$ to the value being written and continue to the next statement.

This part of the pseudocode

```
e3: num[i] :=
      e2:   any nat > maximum(num[1], ..., num[N]);
```

This part of the pseudocode

```
e3: num[i] :=
      e2:   any nat > maximum(num[1], ..., num[N]);
```

The process first evaluates this expression.

This part of the pseudocode

```
e3: num[i] :=
      e2:   any nat > maximum(num[1], ..., num[N]);
```

represented by

```
or    { flag[self] := TRUE;

        } ;
```

$e2$:

$e3$:

This part of the pseudocode

```
e3: num[i] :=
    e2:   any nat > maximum(num[1], ..., num[N]);
```

represented by

or { $flag[self] := \text{TRUE}$;
$unchecked := Procs \setminus \{self\}$;

} ;

$e2$:

$e3$:

This part of the pseudocode

```
e3: num[i] :=
      e2:   any nat > maximum(num[1], ..., num[N]);
```

represented by

or { $flag[self] := \text{TRUE}$;
$unchecked := Procs \setminus \{self\}$;
} ;

The set of processes whose numbers haven't yet been read.

$e2$:

$e3$:

This part of the pseudocode

```
e3: num[i] :=
    e2:   any nat > maximum(num[1], ..., num[N]);
```

represented by

or { $flag[self] := \text{TRUE}$;
$unchecked := Procs \setminus \{self\}$; } ;

$num[self] = 0$, so it doesn't have to be read.

$e2$:

$e3$:

This part of the pseudocode

```
e3: num[i] :=
    e2:   any nat > maximum(num[1], ..., num[N]);
```

represented by

or { $flag[self] := \text{TRUE}$;
$unchecked := Procs \setminus \{self\}$;
$max := 0$
} ;

$e2$:

$e3$:

This part of the pseudocode

```
e3: num[i] :=
      e2:   any nat > maximum(num[1], ..., num[N]);
```

represented by

or { $flag[self] := \text{TRUE}$;
$unchecked := Procs \setminus \{self\}$;
$max := 0$ The largest number read so far.
} ;

$e2$:

$e3$:

This part of the pseudocode

```
e3: num[i] :=
      e2:   any nat > maximum(num[1], ..., num[N]);
```

represented by

or { $flag[self] := \text{TRUE}$;
$unchecked := Procs \setminus \{self\}$;
$max := 0$
} ;

$e2$:

$e3$:

$unchecked$ and max are local variables, so they can be accessed in any atomic action.

This part of the pseudocode

```
e3: num[i] :=
      e2:   any nat > maximum(num[1], ..., num[N]);
```

represented by

```
      or      { flag[self] := TRUE;
                unchecked := Procs \ {self};
                max := 0
              } ;
e2:   while ( unchecked ≠ {} ) Loop until unchecked empty.
        {




        } ;
e3:
```

This part of the pseudocode

```
e3: num[i] :=
      e2:   any nat > maximum(num[1], ..., num[N]);
```

represented by

```
      or      { flag[self] := TRUE;
                unchecked := Procs \ {self};
                max := 0
              } ;
e2:   while ( unchecked ≠ {} )
        { with ( i ∈ unchecked )
            { Locally sets i to a nondeterministically
              chosen element of unchecked .
            }
        } ;
e3:
```

This part of the pseudocode

```
e3: num[i] :=
      e2:   any nat > maximum(num[1], ..., num[N]);
```

represented by

```
      or      { flag[self] := TRUE;
                unchecked := Procs \ {self};
                max := 0
              } ;
e2:   while ( unchecked ≠ {} )
        { with ( i ∈ unchecked )
            { unchecked := unchecked \ {i};
                  Remove process i from unchecked .
            }
        } ;
e3:
```

This part of the pseudocode

```
e3: num[i] :=
      e2:   any nat > maximum(num[1], ..., num[N]);
```

represented by

```
      or      { flag[self] := TRUE;
                unchecked := Procs \ {self};
                max := 0
              } ;
e2:   while ( unchecked ≠ {} )
        { with ( i ∈ unchecked )
            { unchecked := unchecked \ {i};
              if ( num[i] > max ) { max := num[i] }
            }
        } ;
e3:
```

This part of the pseudocode

```
e3: num[i] :=
      e2:   any Nat > maximum(num[1], ..., num[N]);
```

This part of the pseudocode

```
e3: num[i] :=
      e2:   any Nat > maximum(num[1], ..., num[N]);
```

represented by

$e3$: **either** { }
or { } ;

This part of the pseudocode

```
e3: num[i] :=
      e2:   any Nat > maximum(num[1], ..., num[N]);
```

represented by

```
e3:  either {
                    }
     or     {

             } ;
```

Another safe register assignment.

This part of the pseudocode

```
e3: num[i] :=
      e2:   any Nat > maximum(num[1], ..., num[N]);
```

represented by

$e3$: **either { with ($k \in Nat$) { $num[self] := k$ } ;**
}
or {

} ;

either set $num[self]$ to any element of Nat

This part of the pseudocode

```
e3: num[i] :=
      e2:   any Nat > maximum(num[1], ..., num[N]);
```

represented by

$e3$: **either** { **with** ($k \in Nat$) { $num[self] := k$ } ;
goto $e3$ }
or {

} ;

either set $num[self]$ to any element of Nat, and repeat.

This part of the pseudocode

```
e3: num[i] :=
      e2:   any Nat > maximum(num[1], ..., num[N]);
```

represented by

$e3$: **either** { **with** ($k \in Nat$) { $num[self] := k$ } ;
goto $e3$ }
or { **with** ($i \in \{j \in Nat : j > max\}$)
{ $num[self] := i$ }
} ;

either set $num[self]$ to any element of Nat, and repeat.

or

This part of the pseudocode

```
e3: num[i] :=
      e2:   any Nat > maximum(num[1], ..., num[N]);
```

represented by

$e3$: **either** { **with** ($k \in Nat$) { $num[self] := k$ } ;
goto $e3$ }
or { **with** ($i \in \{j \in Nat : j > max\}$)
{ $num[self] := i$ }
} ;

either set $num[self]$ to any element of Nat, and repeat.

or set $num[self]$ to a nondeterministically chosen i in

This part of the pseudocode

```
e3: num[i] :=
      e2:   any Nat > maximum(num[1], ..., num[N]);
```

represented by

$e3$: **either** { **with** ($k \in Nat$) { $num[self] := k$ } ;
goto $e3$ }
or { **with** ($i \in \{j \in Nat : j > max\}$)
{ $num[self] := i$ }
} ;

either set $num[self]$ to any element of Nat, and repeat.

or set $num[self]$ to a nondeterministically chosen i in the set of all numbers in Nat greater than max .

This part of the pseudocode

```
e3: num[i] :=          max equals
      e2:   any Nat > maximum(num[1], ..., num[N]);
```

represented by

$e3$: **either** { **with** ($k \in Nat$) { $num[self] := k$ } ;
goto $e3$ }
or { **with** ($i \in \{j \in Nat : j > max\}$)
{ $num[self] := i$ }
} ;

either set $num[self]$ to any element of Nat, and repeat.

or set $num[self]$ to a nondeterministically chosen i in the set of all numbers in Nat greater than max .

This part of the pseudocode

```
e4: flag[i] := false;
```

This part of the pseudocode

```
e1: flag[i] := true;
```

Just like previous assignment to $flag[i]$

This part of the pseudocode

```
e4: flag[i] := false;
```

represented by

$e4$: **either** { $flag[self] := \neg flag[self]$;
goto $e4$ }
or { $flag[self] :=$ FALSE ;

} ;

This part of the pseudocode

```
for all j in {1,...,N} except j=i do
  begin
w1:


  end;
```

This part of the pseudocode

```
for all j in {1,...,N} except j=i do
  begin
w1:


  end;
```

represented by

```
      or      { flag[self] := FALSE ;

               } ;
w1:   while (                  )
        {




       } ;
```

This part of the pseudocode

```
for all j in {1,...,N} except j=i do
  begin
w1:


  end;
```

represented by

or { $flag[self] := \text{FALSE}$;
$unchecked := Procs \setminus \{self\}$
} ;

$w1$: **while** ( ) {

} ;

Set $unchecked$ to the set of all processes except $self$.

This part of the pseudocode

```
for all j in {1, ..., N} except j=i do
  begin
w1:


  end;
```

represented by

$$
\begin{array}{ll}
 & \textbf{or} \quad \{\ \mathit{flag}[\mathit{self}] := \text{FALSE}\ ; \\
 & \qquad\ \ \mathit{unchecked} := \mathit{Procs} \setminus \{\mathit{self}\} \\
 & \qquad \}\ ; \\
w1: & \textbf{while}\ (\ \mathit{unchecked} \neq \{\}\ ) \\
 & \ \{ \\
 & \\
 & \\
 & \\
 & \\
 & \\
 & \}\ ;
\end{array}
$$

While $\mathit{unchecked}$ is not the empty set

This part of the pseudocode

```
for all j in {1, ..., N} except j=i do
  begin
w1:


  end;
```

represented by

$$
\begin{array}{ll}
 & \textbf{or} \quad \{\ \mathit{flag}[\mathit{self}] := \text{FALSE}\ ; \\
 & \qquad\ \ \mathit{unchecked} := \mathit{Procs} \setminus \{\mathit{self}\} \\
 & \qquad \}\ ; \\
w1: & \textbf{while}\ (\ \mathit{unchecked} \neq \{\}\ ) \\
 & \quad \{ \quad \textbf{with}\ (\ i \in \mathit{unchecked}\ )\ \{\ \mathit{nxt} := i\ \}\ ; \\
 & \qquad\qquad \text{Set } \mathit{nxt} \text{ to a nondeterministically} \\
 & \qquad\qquad \text{chosen } i \text{ in } \mathit{unchecked}\,. \\
 & \quad \}\ ;
\end{array}
$$

This part of the pseudocode

```
for all j in {1, ..., N} except j=i do
  begin
w1:


  end;
```

represented by

$$
\begin{array}{ll}
 & \textbf{or} \quad \{\ \mathit{flag}[\mathit{self}] := \text{FALSE}\ ; \\
 & \qquad\quad \mathit{unchecked} := \mathit{Procs} \setminus \{\mathit{self}\} \\
 & \qquad \}\ ; \\
w1: & \textbf{while}\ (\ \mathit{unchecked} \neq \{\}\ ) \\
 & \quad \{ \quad \textbf{with}\ (\ i \in \mathit{unchecked}\ )\ \{\ \mathit{nxt} := i\ \}\ ; \\
 & \qquad\quad \mathit{unchecked} := \mathit{unchecked} \setminus \{\mathit{nxt}\}\ ; \\
 & \quad \}\ ;
\end{array}
$$

Remove nxt from $\mathit{unchecked}$.

This part of the pseudocode

```
for all j in {1,...,N} except j=i do
  begin
w1:  wait for not flag[j];
w2: wait for num[j] = 0 or
                (num[i], i) < (num[j], j);
  end;
```

This part of the pseudocode

```
for all j in {1, ..., N} except j=i do
  begin
  w1:  wait for not flag[j];
  w2: wait for num[j] = 0 or
                  (num[i], i) < (num[j], j);
  end;
```

represented by

$$
\begin{array}{l}
\quad\quad\quad\quad \mathit{unchecked} := \mathit{Procs} \setminus \{\mathit{self}\} \\
\quad\quad\quad \} \ ; \\
w1: \ \textbf{while} \ (\ \mathit{unchecked} \neq \{\} \ ) \\
\quad \{ \quad \textbf{with} \ (\ i \in \mathit{unchecked} \ ) \ \{ \ \mathit{nxt} := i \ \} \ ; \\
\quad\quad\quad \textbf{await} \ \neg \mathit{flag}[\mathit{nxt}] \, ; \\
\quad w2: \textbf{await} \quad \mathit{num}[\mathit{nxt}] = 0 \\
\quad\quad\quad\quad\quad \vee \langle \mathit{num}[\mathit{self}], \mathit{self} \rangle \prec \langle \mathit{num}[\mathit{nxt}], \mathit{nxt} \rangle \, ; \\
\quad\quad\quad \mathit{unchecked} := \mathit{unchecked} \setminus \{\mathit{nxt}\} \, ; \\
\quad \} \ ;
\end{array}
$$

This part of the pseudocode

```
for all j in {1, ..., N} except j=i do
  begin
w1: wait for not flag[j];
w2: wait for num[j] = 0 or
                (num[i], i) < (num[j], j);
  end;
```

represented by

$$
\begin{array}{ll}
 & \quad unchecked := Procs \setminus \{self\} \\
 & \} \ ; \\
w1: & \textbf{while} \ (\ unchecked \neq \{\}\ ) \\
 & \{ \quad \textbf{with} \ (\ i \in unchecked\ ) \ \{\ nxt := i\ \}\ ; \\
 & \quad \textbf{await} \ \neg flag[nxt]\ ; \\
w2: & \quad \textbf{await} \quad num[nxt] = 0 \\
 & \qquad\qquad \vee \langle num[self],\ self\rangle \prec \langle num[nxt],\ nxt\rangle\ ; \\
 & \quad unchecked := unchecked \setminus \{nxt\}\ ; \\
 & \}\ ;
\end{array}
$$

This part of the pseudocode

```
for all j in {1, ..., N} except j=i do
  begin
w1:  wait for not flag[j];
w2: wait for num[j] = 0 or
                (num[i], i) < (num[j], j);
  end;
```

represented by

$$
\begin{array}{l}
\quad\quad\quad\quad unchecked := Procs \setminus \{self\} \\
\quad\quad\quad\quad \} \ ; \\
w1\text{:}\ \ \textbf{while}\ (\ unchecked \neq \{\}\ ) \\
\quad\quad \{\quad \textbf{with}\ (\ i \in unchecked\ )\ \{\ nxt := i\ \}\ ; \\
\quad\quad\quad\quad \textbf{await}\ \neg flag[nxt]\ ; \\
\quad\quad\quad\quad w2\text{:}\ \textbf{await}\quad num[nxt] = 0 \\
\quad\quad\quad\quad\quad\quad\quad\quad \vee \langle num[self],\ self\rangle \prec \langle num[nxt],\ nxt\rangle\ ; \\
\quad\quad\quad\quad unchecked := unchecked \setminus \{nxt\}\ ; \\
\quad\quad \}\ ;
\end{array}
$$

The rest of the pseudocode

```
    cs: critical section;
  exit: num[i] := 0;
end
```

The rest of the pseudocode

```
      cs: critical section;
    exit: num[i] := 0;
  end
```

represented by

$$
\begin{array}{ll}
cs: & \textbf{skip}\,; \\
exit: & \textbf{either}\ \{\ \textbf{with}\ (\ k \in Nat\ )\ \{\ num[self] := k\ \}\ ; \\
 & \qquad\ \ \textbf{goto}\ exit\ \} \\
 & \textbf{or}\quad\ \{\ num[self] := 0\ \}
\end{array}
$$

}

The rest of the pseudocode

```
      cs: critical section;
    exit: num[i] := 0;
  end
```

represented by

$$
\begin{array}{ll}
cs: & \textbf{skip}\ ; \\
exit: & \textbf{either}\ \{\ \textbf{with}\ (\ k \in Nat\ )\ \{\ num[self] := k\ \}\ ; \\
 & \quad\ \ \textbf{goto}\ exit\ \} \\
 & \textbf{or}\ \ \ \ \{\ num[self] := 0\ \} \\
\}
\end{array}
$$

The rest of the pseudocode

```
      cs: critical section;
    exit: num[i] := 0;
  end
```

represented by

$$
\begin{array}{ll}
cs: & \textbf{skip}\,; \\
exit: & \textbf{either}\ \{\ \textbf{with}\ (\ k \in Nat\ )\ \{\ num[self] := k\ \}\ ; \\
 & \qquad\ \textbf{goto}\ exit\ \} \\
 & \textbf{or}\quad\ \{\ num[self] := 0\ \}
\end{array}
$$

}

Another safe register assignment.

The “Complete” Pseudocode

The “Complete” Pseudocode

Omits:

- the initialization of variables
- the declarations of N, i, and the set of processes
- the definition of $<$ on pairs
- the semantics of read/write for safe registers

The “Complete” Pseudocode

Omits:

- the initialization of variables
- the declarations of N, i, and the set of processes
- the definition of $<$ on pairs
- the semantics of read/write for safe registers

They’re written in prose.

The “Complete” Pseudocode

```
while true do
  begin
    ncs: noncritical section;
     e1: flag[i] := true;
     e3: num[i] :=
          e2: any nat > maximum(num[1], ..., num[N]);
     e4: flag[i] := false;
         for all j in {1, ..., N} except j=i do
           begin
         w1: wait for not flag[j];
         w2: wait for num[j] = 0 or
                        (num[i], i) < (num[j], j);
           end;
     cs: critical section;
   exit: num[i] := 0;
  end
```

The “Complete” PlusCal Code

The "Complete" PlusCal Code

Omits:

- the declaration of N
- the definitions of $Procs$ and $\prec$
- a little boilerplate

The "Complete" PlusCal Code

Omits:

- the declaration of N
- the definitions of $Procs$ and $\prec$
- a little boilerplate

They're in the TLA$^+$ module.

The "Complete" PlusCal Code

```
--algorithm Bakery
{ variables num = [i ∈ Procs ↦ 0], flag = [i ∈ Procs ↦ FALSE] ;
  process ( p ∈ Procs )
    variables unchecked = {}, max = 0, nxt = 1 ;
    { ncs: while ( TRUE )
           { e1:  either { flag[self] := ¬flag[self] ;
                           goto e1 }
                  or     { flag[self] := TRUE ;
                           unchecked := Procs \ {self} ;
                           max := 0
                         } ;
             e2:  while ( unchecked ≠ {} )
                    { with ( i ∈ unchecked )
                        { unchecked := unchecked \ {i} ;
                          if ( num[i] > max ) { max := num[i] }
                        }
                    } ;
             e3:  either { with ( k ∈ Nat ) { num[self] := k } ;
                           goto e3 }
                  or     { with ( i ∈ {j ∈ Nat : j > max} )
                             { num[self] := i }
                         } ;
             e4:  either { flag[self] := ¬flag[self] ;
                           goto e4 }
                  or     { flag[self] := FALSE ;
                           unchecked := Procs \ {self}
                         } ;
             w1:  while ( unchecked ≠ {} )
                    {    with ( i ∈ unchecked ) { nxt := i } ;
                         await ¬flag[nxt] ;
                    w2:  await  ∨ num[nxt] = 0
                                ∨ ⟨num[self], self⟩ ≺ ⟨num[nxt], nxt⟩ ;
                         unchecked := unchecked \ {nxt} ;
                    } ;
             cs:  skip ;
             exit: either { with ( k ∈ Nat ) { num[self] := k } ;
                            goto exit }
                   or     { num[self] := 0 }
           }
    }
}
```

The TLA$^+$ Translation

The TLA$^+$ Translation

The translator puts the TLA$^+$ version of the algorithm in the module.

The TLA$^+$ Translation

The translator puts the TLA$^+$ version of the algorithm in the module.

This version is the mathematical semantics of the algorithm

The TLA$^+$ Translation

The translator puts the TLA$^+$ version of the algorithm in the module.

This version is the mathematical semantics of the algorithm, and it's easy to understand.

The TLA$^+$ Translation

The translator puts the TLA$^+$ version of the algorithm in the module.

This version is the mathematical semantics of the algorithm, and it's easy to understand.

The translation introduces a variable pc to describe the control state.

The TLA$^+$ Translation

The translator puts the TLA$^+$ version of the algorithm in the module.

This version is the mathematical semantics of the algorithm, and it's easy to understand.

The translation introduces a variable pc to describe the control state.

Most of the translation consists of:

for each label, a formula describing the atomic step that starts at the label.

A Trivial Example

PlusCal:

cs: **skip ;**
$exit$:

A Trivial Example

PlusCal:

cs: **skip ;**
$exit$:

TLA$^+$ Translation:

A Trivial Example

PlusCal:

cs: **skip ;**
$exit$:

TLA$^+$ Translation:

$cs(self) \triangleq$

Defines the formula describing the execution of this atomic step by a process named $self$.

A Trivial Example

PlusCal:

cs: **skip ;**
$exit$:

TLA$^+$ Translation:

$$cs(self) \triangleq \begin{array}{l} \land \\ \land \\ \land \end{array}$$

The conjunction of three formulas.

A Trivial Example

PlusCal:

cs: **skip ;**
$exit$:

TLA$^+$ Translation:

$$cs(self) \triangleq \land \land \land$$

This bulleted list notation uses indentation to eliminate parentheses, which is great for large formulas.

A Trivial Example

PlusCal:

cs: **skip ;**
$exit$:

TLA$^+$ Translation:

$$cs(self) \triangleq \land\, pc[self] = \text{“cs”} \land \land$$

True iff control in process $self$ is at cs .

A Trivial Example

PlusCal:

cs: **skip ;**
$exit$:

TLA$^+$ Translation:

$$cs(self) \triangleq \land\ pc[self] = \text{“cs”} \land \land$$

True iff control in process $self$ is at cs .

A condition on the first state of the step (a precondition).

A Trivial Example

PlusCal:

cs: **skip ;**
$exit$:

TLA$^+$ Translation:

$$cs(self) \triangleq \land pc[self] = \text{“cs”} \land pc' = \land$$

Describes the value of pc after the step.

A Trivial Example

PlusCal:

cs: **skip ;**
$exit$:

TLA$^+$ Translation:

$$cs(self) \triangleq \land\ pc[self] = \text{“cs”} \\ \land\ pc' = [pc \text{ EXCEPT } ![self] = \text{“exit”}] \\ \land$$

TLA$^+$ notation for the value of the function / array pc after executing $pc[self] := \text{“exit”}$.

A Trivial Example

PlusCal:

cs: **skip ;**
$exit$:

TLA$^+$ Translation:

$$cs(self) \triangleq \begin{array}{l} \land\ pc[self] = \text{“cs”} \\ \land\ pc' = [pc \text{ EXCEPT } ![self] = \text{“exit”}] \\ \land\ \text{UNCHANGED } \langle num, flag, unchecked, max, nxt \rangle \end{array}$$

Asserts that values of the variables num , . . . , nxt are not changed.

A Trivial Example

PlusCal:

cs: **skip ;**
$exit$:

TLA$^+$ Translation:

$$cs(self) \triangleq \begin{array}{l} \land\ pc[self] = \text{“cs”} \\ \land\ pc' = [pc \text{ EXCEPT } ![self] = \text{“exit”}] \\ \land\ \text{UNCHANGED } \langle num, flag, unchecked, max, nxt \rangle \end{array}$$

You have now seen about all the TLA$^+$ notation needed to understand the translation.

A Typical Example

PlusCal:

$w1$: **while** ($unchecked \neq \{\}$)
{ **with** ($i \in unchecked$) { $nxt := i$ } ;
await $\neg flag[nxt]$;
$w2$:
} ;
cs:

A Typical Example

PlusCal:

$w1$: **while** ($unchecked \neq \{\}$)
{ **with** ($i \in unchecked$) { $nxt := i$ } ;
await $\neg flag[nxt]$;
$w2$:
} ;
cs:

TLA+ Translation:

$$
\begin{aligned}
w1(self) \triangleq\ & \land\ pc[self] = \text{“w1”} \\
& \land\ \text{IF}\ unchecked[self] \neq \{\} \\
& \qquad \text{THEN}\ \land \exists\, i \in unchecked[self] : \\
& \qquad\qquad\qquad nxt' = [nxt\ \text{EXCEPT}\ ![self] = i] \\
& \qquad\qquad\ \land \neg flag[nxt'[self]] \\
& \qquad\qquad\ \land\ pc' = [pc\ \text{EXCEPT}\ ![self] = \text{“w2”}] \\
& \qquad \text{ELSE}\ \ \land\ pc' = [pc\ \text{EXCEPT}\ ![self] = \text{“cs”}] \\
& \qquad\qquad\ \land\ nxt' = nxt \\
& \land\ \text{UNCHANGED}\ \langle num,\ flag,\ unchecked,\ max \rangle
\end{aligned}
$$

A Typical Example

PlusCal:

$w1$: **while** ($unchecked \neq \{\}$)
{ **with** ($i \in unchecked$) { $nxt := i$ } ;
await $\neg flag[nxt]$;
$w2$:
} ;
cs:

TLA+ Translation:

Don't try to read it.

$$
\begin{aligned}
w1(self) \triangleq\ & \land\ pc[self] = \text{“w1”} \\
& \land\ \text{IF}\ unchecked[self] \neq \{\} \\
& \qquad \text{THEN}\ \land \exists\, i \in unchecked[self] : \\
& \qquad\qquad\qquad nxt' = [nxt\ \text{EXCEPT}\ ![self] = i] \\
& \qquad\qquad\ \land \neg flag[nxt'[self]] \\
& \qquad\qquad\ \land\ pc' = [pc\ \text{EXCEPT}\ ![self] = \text{“w2”}] \\
& \qquad \text{ELSE}\ \ \land\ pc' = [pc\ \text{EXCEPT}\ ![self] = \text{“cs”}] \\
& \qquad\qquad\ \land\ nxt' = nxt \\
& \land\ \text{UNCHANGED}\ \langle num, flag, unchecked, max \rangle
\end{aligned}
$$

A Typical Example

PlusCal:

$w1$: **while** ($unchecked \neq \{\}$)
{ **with** ($i \in unchecked$) { $nxt := i$ } ;
await $\neg flag[nxt]$;
$w2$:
} ;
cs:

TLA+ Translation:

Don't try to read it.

I'll just point out one thing.

$$
\begin{array}{lll}
w1(self) \triangleq & \land\ pc[self] = \text{“w1”} & \\
& \land\ \text{IF}\ unchecked[self] \neq \{\} & \\
& \quad \text{THEN} & \land\ \exists\, i \in unchecked[self] : \\
& & \qquad nxt' = [nxt\ \text{EXCEPT}\ ![self] = i] \\
& & \land\ \neg flag[nxt'[self]] \\
& & \land\ pc' = [pc\ \text{EXCEPT}\ ![self] = \text{“w2”}] \\
& \quad \text{ELSE} & \land\ pc' = [pc\ \text{EXCEPT}\ ![self] = \text{“cs”}] \\
& & \land\ nxt' = nxt \\
& \land\ \text{UNCHANGED}\ \langle num, flag, unchecked, max \rangle &
\end{array}
$$

A Typical Example

PlusCal:

$w1$: **while** ($unchecked \neq \{\}$)
{ **with** ($i \in unchecked$) { $nxt := i$ } ;
await $\neg flag[nxt]$;
$w2$:
} ;
cs:

Nondeterministic choice

TLA+ Translation:

$$
\begin{array}{lll}
w1(self) \triangleq & \land\ pc[self] = \text{“w1”} & \\
& \land\ \text{IF}\ unchecked[self] \neq \{\} & \\
& \quad \text{THEN} & \land\ \exists\, i \in unchecked[self] : \\
& & \quad nxt' = [nxt\ \text{EXCEPT}\ ![self] = i] \\
& & \land\ \neg flag[nxt'[self]] \\
& & \land\ pc' = [pc\ \text{EXCEPT}\ ![self] = \text{“w2”}] \\
& \quad \text{ELSE} & \land\ pc' = [pc\ \text{EXCEPT}\ ![self] = \text{“cs”}] \\
& & \land\ nxt' = nxt \\
& \land\ \text{UNCHANGED}\ \langle num, flag, unchecked, max \rangle &
\end{array}
$$

A Typical Example

PlusCal:

$w1$: **while** ($unchecked \neq \{\}$)
{ **with** ($i \in unchecked$) { $nxt := i$ } ;
await $\neg flag[nxt]$;
$w2$:
} ;
cs:

Nondeterministic choice

TLA+ Translation:

$$
\begin{aligned}
w1(self) \triangleq\ & \land\ pc[self] = \text{“w1”} \\
& \land\ \text{IF}\ unchecked[self] \neq \{\} \\
& \quad \text{THEN}\ \land \exists\, i \in unchecked[self] : \\
& \qquad\qquad nxt' = [nxt\ \text{EXCEPT}\ ![self] = i] \\
& \qquad\ \land \neg flag[nxt'[self]] \\
& \qquad\ \land\ pc' = [pc\ \text{EXCEPT}\ ![self] = \text{“w2”}] \\
& \quad \text{ELSE}\ \land\ pc' = [pc\ \text{EXCEPT}\ ![self] = \text{“cs”}] \\
& \qquad\ \land\ nxt' = nxt \\
& \land\ \text{UNCHANGED}\ \langle num, flag, unchecked, max \rangle
\end{aligned}
$$

Is represented mathematically by $\exists$.

Why Should You Use PlusCal?

1. It’s precise

1. It's precise

This publication was widely read:

Paxos Made Simple

ACM SIGACT News

December 2001

1. It's precise

This publication was widely read:

Paxos Made Simple
ACM SIGACT News
December 2001

Not long ago I was told that one sentence in it can be misinterpreted

1. It's precise

This publication was widely read:

Paxos Made Simple
ACM SIGACT News
December 2001

Not long ago I was told that one sentence in it can be misinterpreted, and that led to bugs in two industrial systems.

2. You can model check the algorithm.

2. You can model check the algorithm.

Example: An algorithm published in

The Mailbox Problem

Aguilera, Gafni, and Lamport

DISC 2008

2. You can model check the algorithm.

Example: An algorithm published in

The Mailbox Problem

Aguilera, Gafni, and Lamport

DISC 2008

Our algorithm for finding the algorithm. . .

begin : `Eli sends me an algorithm;`

begin : Eli sends me an algorithm;

I code it in PlusCal;

begin : Eli sends me an algorithm;

I code it in PlusCal;

I run the model checker;

begin : Eli sends me an algorithm;
I code it in PlusCal;
I run the model checker;
if it finds an error

```
begin : Eli sends me an algorithm;
        I code it in PlusCal;
        I run the model checker;
        if it finds an error
           then { I send the error trace to Eli;
```

begin : Eli sends me an algorithm;
I code it in PlusCal;
I run the model checker;
if it finds an error
then { I send the error trace to Eli;
goto *begin* }

```
begin : Eli sends me an algorithm;
        I code it in PlusCal;
        I run the model checker;
        if it finds an error
           then { I send the error trace to Eli;
                  goto begin }
```

We repeated this loop about 6 times.

```
begin : Eli sends me an algorithm;
        I code it in PlusCal;
        I run the model checker;
        if it finds an error
           then { I send the error trace to Eli;
                  goto begin }
        The three of us start thinking hard;
```

```
begin : Eli sends me an algorithm;
        I code it in PlusCal;
        I run the model checker;
        if it finds an error
           then { I send the error trace to Eli;
                  goto begin }
        The three of us start thinking hard;
        We write a rigorous proof
```

Thinking is hard.

Thinking is hard.

Why waste time and brainpower thinking about an algorithm if the model checker can tell you it's wrong.

Thinking is hard.

Why waste time and brainpower thinking about an algorithm if the model checker can tell you it's wrong.

Model check first, think later.

3. You can write a rigorous, machine-checked proof.

3. You can write a rigorous, machine-checked proof.

To prove:

No two processes ever in critical section at the same time.

3. You can write a rigorous, machine-checked proof.

To prove:

No two processes ever in critical section at the same time.

Invariance of $MutualExclusion$, defined by:

$$MutualExclusion \triangleq$$
$$\forall\, i, j \in Procs : (i \neq j) \Rightarrow \neg \wedge pc[i] = \text{“cs”} \wedge pc[j] = \text{“cs”}$$

The Inductive Invariant

$$
\begin{array}{l}
\land\ TypeOK \\
\land\ \forall\, i \in Procs : \\
\qquad \land\ (pc[i] \in \{\text{“ncs”}, \text{“e1”}, \text{“e2”}\}) \Rightarrow (num[i] = 0) \\
\qquad \land\ (pc[i] \in \{\text{“e4”}, \text{“w1”}, \text{“w2”}, \text{“cs”}\}) \Rightarrow (num[i] \neq 0) \\
\qquad \land\ (pc[i] \in \{\text{“e2”}, \text{“e3”}\}) \Rightarrow flag[i] \\
\qquad \land\ (pc[i] = \text{“w2”}) \Rightarrow (nxt[i] \neq i) \\
\qquad \land\ pc[i] \in \{\text{“e2”}, \text{“w1”}, \text{“w2”}\} \Rightarrow i \notin unchecked[i] \\
\qquad \land\ (pc[i] \in \{\text{“w1”}, \text{“w2”}\}) \Rightarrow \\
\qquad\qquad \forall\, j \in (Procs \setminus unchecked[i]) \setminus \{i\} : Before(i, j) \\
\qquad \land\ \land\ (pc[i] = \text{“w2”}) \\
\qquad\qquad \land\ \lor\ (pc[nxt[i]] = \text{“e2”}) \land (i \notin unchecked[nxt[i]]) \\
\qquad\qquad\qquad \lor\ pc[nxt[i]] = \text{“e3”} \\
\qquad\qquad \Rightarrow max[nxt[i]] \geq num[i] \\
\qquad \land\ (pc[i] = \text{“cs”}) \Rightarrow \forall\, j \in Procs \setminus \{i\} : Before(i, j)
\end{array}
$$

The Inductive Invariant

$$
\begin{array}{l}
\land \boxed{TypeOK} \\
\land \forall\, i \in Procs : \\
\quad \land (pc[i] \in \{\text{“ncs”}, \text{“e1”}, \text{“e2”}\}) \Rightarrow (num[i] = 0) \\
\quad \land (pc[i] \in \{\text{“e4”}, \text{“w1”}, \text{“w2”}, \text{“cs”}\}) \Rightarrow (num[i] \neq 0) \\
\quad \land (pc[i] \in \{\text{“e2”}, \text{“e3”}\}) \Rightarrow flag[i] \\
\quad \land (pc[i] = \text{“w2”}) \Rightarrow (nxt[i] \neq i) \\
\quad \land pc[i] \in \{\text{“e2”}, \text{“w1”}, \text{“w2”}\} \Rightarrow i \notin unchecked[i] \\
\quad \land (pc[i] \in \{\text{“w1”}, \text{“w2”}\}) \Rightarrow \\
\qquad \forall\, j \in (Procs \setminus unchecked[i]) \setminus \{i\} : Before(i, j) \\
\quad \land \land (pc[i] = \text{“w2”}) \\
\qquad \land \lor (pc[nxt[i]] = \text{“e2”}) \land (i \notin unchecked[nxt[i]]) \\
\qquad\quad \lor pc[nxt[i]] = \text{“e3”} \\
\qquad \Rightarrow max[nxt[i]] \geq num[i] \\
\quad \land (pc[i] = \text{“cs”}) \Rightarrow \forall\, j \in Procs \setminus \{i\} : Before(i, j)
\end{array}
$$

A simple formula asserting type correctness:
the value of each variable is an element of the right set.

The Inductive Invariant

$$
\begin{array}{l}
\land\ TypeOK \\
\land\ \forall\, i \in Procs : \\
\quad \land\ (pc[i] \in \{\text{“ncs”}, \text{“e1”}, \text{“e2”}\}) \Rightarrow (num[i] = 0) \\
\quad \land\ (pc[i] \in \{\text{“e4”}, \text{“w1”}, \text{“w2”}, \text{“cs”}\}) \Rightarrow (num[i] \neq 0) \\
\quad \land\ (pc[i] \in \{\text{“e2”}, \text{“e3”}\}) \Rightarrow flag[i] \\
\quad \land\ (pc[i] = \text{“w2”}) \Rightarrow (nxt[i] \neq i) \\
\quad \land\ pc[i] \in \{\text{“e2”}, \text{“w1”}, \text{“w2”}\} \Rightarrow i \notin unchecked[i] \\
\quad \land\ (pc[i] \in \{\text{“w1”}, \text{“w2”}\}) \Rightarrow \\
\qquad \forall\, j \in (Procs \setminus unchecked[i]) \setminus \{i\} : \boxed{Before(i, j)} \\
\quad \land\ \land\ (pc[i] = \text{“w2”}) \\
\qquad \land\ \lor\ (pc[nxt[i]] = \text{“e2”}) \land (i \notin unchecked[nxt[i]]) \\
\qquad\quad \lor\ pc[nxt[i]] = \text{“e3”} \\
\qquad \Rightarrow max[nxt[i]] \geq num[i] \\
\quad \land\ (pc[i] = \text{“cs”}) \Rightarrow \forall\, j \in Procs \setminus \{i\} : \boxed{Before(i, j)}
\end{array}
$$

A formula implying that process i must enter the critical section before process j does.

$$
\begin{array}{lll}
Before(i,\, j) & \triangleq & \land\ num[i] > 0 \\
& & \land\ \lor\ pc[j] \in \{\text{“ncs”},\ \text{“e1”},\ \text{“exit”}\} \\
& & \quad \lor\ \land\ pc[j] = \text{“e2”} \\
& & \qquad \land\ \lor\ i \in unchecked[j] \\
& & \qquad\quad \lor\ max[j] \geq num[i] \\
& & \quad \lor\ \land\ pc[j] = \text{“e3”} \\
& & \qquad \land\ max[j] \geq num[i] \\
& & \quad \lor\ \land\ pc[j] \in \{\text{“e4”},\ \text{“w1”},\ \text{“w2”}\} \\
& & \qquad \land\ \langle num[i],\, i\rangle \prec \langle num[j],\, j\rangle \\
& & \qquad \land\ (pc[j] \in \{\text{“w1”},\ \text{“w2”}\}) \Rightarrow (i \in unchecked[j])
\end{array}
$$

The correctness theorem

THEOREM $Spec \Rightarrow \Box MutualExclusion$

The correctness theorem and its machine-checked proof

THEOREM $Spec \Rightarrow \Box MutualExclusion$
$\langle 1 \rangle$ USE $N \in Nat$ DEFS $Procs$, Inv, $TypeOK$, $Before$, $\prec$, $ProcSet$
$\langle 1 \rangle 1$. $Init \Rightarrow Inv$
BY SMT DEF $Init$
$\langle 1 \rangle 2$. $Inv \wedge [Next]_{vars} \Rightarrow Inv'$
BY $Z3$ DEF $Next$, ncs, p, $e1$, $e2$, $e3$, $e4$, $w1$, $w2$, cs, $exit$, $vars$
$\langle 1 \rangle 3$. $Inv \Rightarrow MutualExclusion$
BY SMT DEF $MutualExclusion$
$\langle 1 \rangle 4$. QED
BY $\langle 1 \rangle 1$, $\langle 1 \rangle 2$, $\langle 1 \rangle 3$, PTL DEF $Spec$

The correctness theorem and its machine-checked proof

THEOREM $Spec \Rightarrow \Box MutualExclusion$
$\langle 1 \rangle$ USE $N \in Nat$ DEFS $Procs$, Inv, $TypeOK$, $Before$, $\prec$, $ProcSet$
$\langle 1 \rangle 1$. $Init \Rightarrow Inv$
BY SMT DEF $Init$
$\langle 1 \rangle 2$. $Inv \wedge [Next]_{vars} \Rightarrow Inv'$
BY $Z3$ DEF $Next$, ncs, p, $e1$, $e2$, $e3$, $e4$, $w1$, $w2$, cs, $exit$, $vars$
$\langle 1 \rangle 3$. $Inv \Rightarrow MutualExclusion$
BY SMT DEF $MutualExclusion$
$\langle 1 \rangle 4$. QED
BY $\langle 1 \rangle 1$, $\langle 1 \rangle 2$, $\langle 1 \rangle 3$, PTL DEF $Spec$

Proofs are seldom this short.

The correctness theorem and its machine-checked proof

THEOREM $Spec \Rightarrow \Box MutualExclusion$
$\langle 1 \rangle$ USE $N \in Nat$ DEFS $Procs$, Inv, $TypeOK$, $Before$, $\prec$, $ProcSet$
$\langle 1 \rangle 1$. $Init \Rightarrow Inv$
BY SMT DEF $Init$
$\langle 1 \rangle 2$. $Inv \wedge [Next]_{vars} \Rightarrow Inv'$
BY $Z3$ DEF $Next$, ncs, p, $e1$, $e2$, $e3$, $e4$, $w1$, $w2$, cs, $exit$, $vars$
$\langle 1 \rangle 3$. $Inv \Rightarrow MutualExclusion$
BY SMT DEF $MutualExclusion$
$\langle 1 \rangle 4$. QED
BY $\langle 1 \rangle 1$, $\langle 1 \rangle 2$, $\langle 1 \rangle 3$, PTL DEF $Spec$

Proofs are seldom this short.

This is a best-case scenario because the proof uses only simple properties of integers and sets of integers.

A More Recent Algorithm

A More Recent Algorithm

Adaptive Register Allocation with a Linear Number of Registers

Delporte-Gallet, Fauconnier, Gafni, and Lamport

DISC 2013

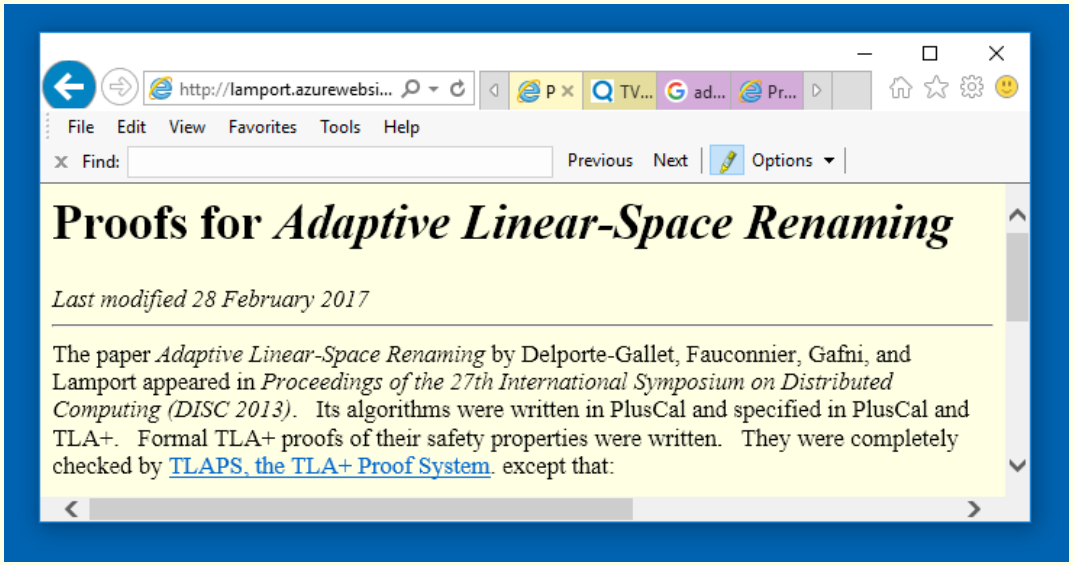
A More Recent Algorithm

Adaptive Register Allocation with a Linear Number of Registers

Delporte-Gallet, Fauconnier, Gafni, and Lamport

DISC 2013

Search the web for *adaptive register pluscal* to find:



The Main Algorithm

```
--algorithm SnapShot
{  variables  result = [p ∈ Proc ↦ {}],
              A2   = [i ∈ Nat ↦ {}],
              A3   = [i ∈ Nat ↦ {}],
    process ( Pr ∈ Proc )
    variables myVals = {},
              known = {},
              notKnown = {},
              lnbpart = 0,
              nbpart = 0,
              nextout = {},
              out = {};
    { a: with ( P ∈ {Q ∈ SUBSET Proc :
                        ∧ self ∈ Q
                        ∧ ∀ p ∈ Proc \ {self} :
                            ∨ Cardinality(result[p]) ≠ Cardinality(Q)
                            ∨ Q = result[p]
                    } )
        { result[self] := P } ;
      A2[Cardinality(result[self]) − 1] := result[self] ;
      b: while ( TRUE )
          {   with ( v ∈ Val ) { myVals := myVals ∪ {v} } ;
              known := myVals ∪ known ;
              nbpart := Cardinality(NUnion(A2)) ;

          c:  lnbpart := nbpart ;
              known := known ∪ NUnion(A3) ;
              notKnown := {i ∈ 0 .. (nbpart − 1) : known ≠ A3[i]} ;
              if ( notKnown ≠ {} ) { d: with ( i ∈ notKnown )
                                          { A3[i] := known } ;
                                     goto c
                                   }
              else if ( nbpart = Cardinality(NUnion(A2)) )
                     { nextout := known } ;

          e:  nbpart := Cardinality(NUnion(A2)) ;
              if ( lnbpart = nbpart ) { out := known }
              else { goto c }
          }
    }
}
```

The machine-checked proof is 1752 lines.

The machine-checked proof is 1752 lines.

Writing machine-checkable proofs is hard.

The machine-checked proof is 1752 lines.

Writing machine-checkable proofs is hard.

Most of you will never write one.

The machine-checked proof is 1752 lines.

Writing machine-checkable proofs is hard.

Most of you will never write one.

So, why use PlusCal?

1. It’s precise

1. It's precise

The mathematical representation of the algorithm permits more rigorous hand proofs.

1. It's precise

The mathematical representation of the algorithm permits more rigorous hand proofs.

And more rigorous hand proofs are more likely to be correct.

2. You can model check the algorithm.

2. You can model check the algorithm.

Model checking can catch errors in a proof.

2. You can model check the algorithm.

Model checking can catch errors in a proof.

The Wikipedia page for Peterson's algorithm used to state that the algorithm worked because it maintained the invariance of a certain predicate.

2. You can model check the algorithm.

Model checking can catch errors in a proof.

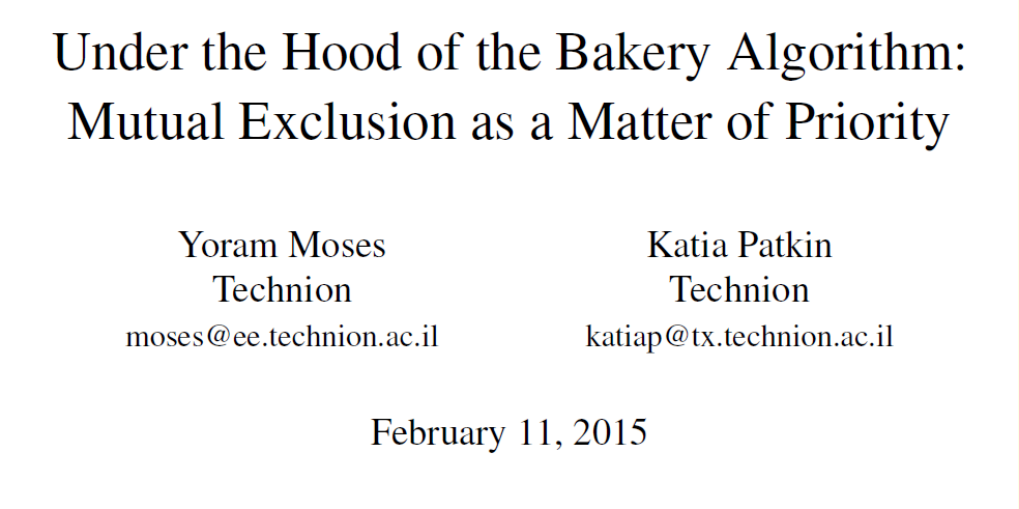
The Wikipedia page for Peterson's algorithm used to state that the algorithm worked because it maintained the invariance of a certain predicate.

Model checking revealed that the predicate was not an invariant.

What led me back to the bakery algorithm in 2015?

What led me back to the bakery algorithm in 2015?

In February 2015, I received this paper from Yoram Moses.



Under the Hood of the Bakery Algorithm:
Mutual Exclusion as a Matter of Priority

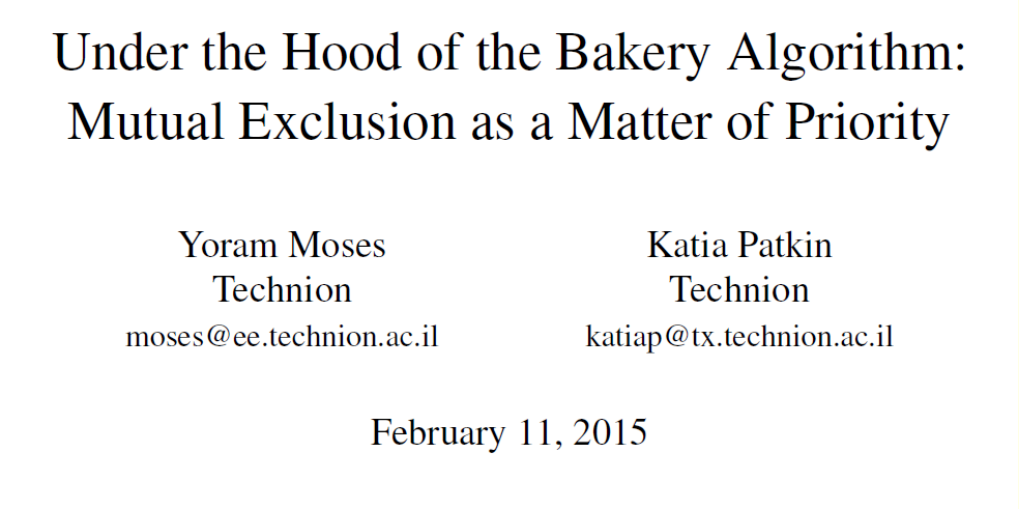
Yoram Moses
Technion
moses@ee.technion.ac.il

Katia Patkin
Technion
katiap@tx.technion.ac.il

February 11, 2015

What led me back to the bakery algorithm in 2015?

In February 2015, I received this paper from Yoram Moses.



Under the Hood of the Bakery Algorithm:
Mutual Exclusion as a Matter of Priority

Yoram Moses
Technion
moses@ee.technion.ac.il

Katia Patkin
Technion
katiap@tx.technion.ac.il

February 11, 2015

It introduces the *boulangerie* algorithm, a variant of the bakery algorithm.

I had already written this spec and proof of the bakery algorithm for other reasons.

I had already written this spec and proof of the bakery algorithm for other reasons.

So, I decided to modify them for the boulangerie algorithm.

The Results

The Results

	bakery	boulangerie	
PlusCal	41	50	lines of “code”
TLA$^+$ translation	99	109	
invariant	31	42	
proof	9	72	

The Results

	bakery	boulangerie
PlusCal	41	50
TLA$^+$ translation	99	109
invariant	31	42
proof	9	72

I typed only about 20 of those lines of proof.

The Results

	bakery	boulangerie
PlusCal	41	50
TLA$^+$ translation	99	109
invariant	31	42
proof	9	72

I typed only about 20 of those lines of proof.

The rest were generated with the Toolbox's *decompose proof* command.

The Results

	bakery	boulangerie
PlusCal	41	50
TLA$^+$ translation	99	109
invariant	31	42
proof	9	72

Transforming *bakery* to *boulangerie* took me less than 4 hours

The Results

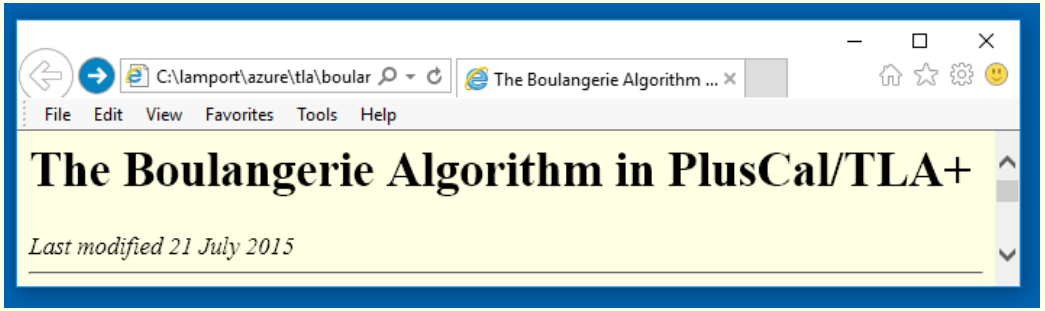
	bakery	boulangerie
PlusCal	41	50
TLA$^+$ translation	99	109
invariant	31	42
proof	9	72

Transforming *bakery* to *boulangerie* took me less than 4 hours (spread over two days).

Find it on the Web

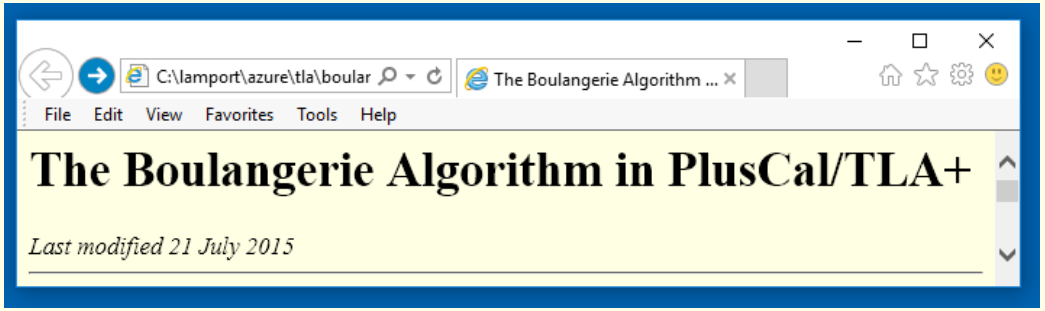
Find it on the Web

The fully commented specs are on this web page:



Find it on the Web

The fully commented specs are on this web page:



To find it, search the web for *boulangerie pluscal*.

Michel:

Michel:

Your colleagues are too busy teaching and being on committees

Michel:

Your colleagues are too busy teaching and being on committees to learn something new.

Michel:

Your colleagues are too busy teaching and being on committees to learn something new.

Most of them are still writing algorithms the way I did in 1979.

Michel:

Your colleagues are too busy teaching and being on committees to learn something new.

Most of them are still writing algorithms the way I did in 1979.

Now you'll have time to venture into the 21st century.

Michel:

Your colleagues are too busy teaching and being on committees to learn something new.

Most of them are still writing algorithms the way I did in 1979.

Now you'll have time to venture into the 21st century.

But whatever you decide to do . . .

Happy Retirement !