A further application of FASE: study liveness properties on Mutual Exclusion Algorithms

speaker: Massimo Callisto De Donato

{massimo.callisto}@unicam.it

Scuola di Scienze e Tecnologie - Università di Camerino
Via Madonna delle Carceri, 9 - 62032 Camerino
http://www.cs.unicam.it

PaCo Meeting - Camerino
September, 15th 2010

Faster Asynchronous Systems Evaluation

FASE (Faster Asynchronous Systems Evaluation):

- Aimed to analyse systems specified in PAFAS algebra
- Different modules loosely-coupled written in Java
- Evaluates the efficiency of a process in the worst-case behaviour
- **Checks the presence of catastrophic cycles**

F. Buti, M. Callisto De Donato, F. Corradini, M. R. Di Berardini and W. Vogler.
Evaluating the Efficiency of Asynchronous Systems with FASE.
In pre-proc. of the 1st Int. Workshop on Quantitative Formal Methods, pp.101-106, Technische Universiteit Eindhoven, 2009

Faster Asynchronous Systems Evaluation

FASE (Faster Asynchronous Systems Evaluation):

- Aimed to analyse systems specified in PAFAS algebra
- Different modules loosely-coupled written in Java
- Evaluates the efficiency of a process in the worst-case behaviour
- **Checks the presence of catastrophic cycles**

Apply FASE to study new properties about systems:

F. Buti, M. Callisto De Donato, F. Corradini, M. R. Di Berardini and W. Vogler.
Evaluating the Efficiency of Asynchronous Systems with FASE.
In pre-proc. of the 1st Int. Workshop on Quantitative Formal Methods, pp.101-106, Technische Universiteit Eindhoven, 2009

Faster Asynchronous Systems Evaluation

FASE (Faster Asynchronous Systems Evaluation):

- Aimed to analyse systems specified in PAFAS algebra
- Different modules loosely-coupled written in Java
- Evaluates the efficiency of a process in the worst-case behaviour
- **Checks the presence of catastrophic cycles**

Apply FASE to study new properties about systems:

- Checks liveness properties
- Under the assumption of fairness of actions
- And a process algebra with **Non-Blocking Reading**
- **By studying the presence of catastrophic cycles**

F. Buti, M. Callisto De Donato, F. Corradini, M. R. Di Berardini and W. Vogler.

Evaluating the Efficiency of Asynchronous Systems with FASE.
In pre-proc. of the 1st Int. Workshop on Quantitative Formal Methods, pp.101-106, Technische Universiteit Eindhoven, 2009

F. Corradini, M.R. Di Berardini, W. Vogler

Time and Fairness in a Process Algebra with Non-Blocking Reading
In Proc. of 35th Conference on Current Trends in Theory and Practice of Computer Science (SOFSEM 2009), LNCS 5404: 193 204 (2009).

Study liveness properties

Liveness

Something good will happen eventually.

Study liveness properties

Liveness

Something good will happen eventually.

Consider a critical resource that must be accessed from concurrent processes in a mutual way. An algorithm is used to regulate the usage of the resource. Different properties can be studied:

- Is the mutual exclusion preserved?
- Is the algorithm **live**?

Study liveness properties

Liveness

Something good will happen eventually.

Consider a critical resource that must be accessed from concurrent processes in a mutual way. An algorithm is used to regulate the usage of the resource. Different properties can be studied:

- Is the mutual exclusion preserved?
- Is the algorithm **live**?

Liveness

Whenever, at some point, a process P_i requests the execution of its critical section, then at some later point it will enter it.

The verification of liveness properties usually requires some fairness assumption.

D.J. Walker

Automated Analysis of Mutual Exclusion algorithms using CCS

Formal Aspects of Computing 1, pp. 273-292, 1989

A characterization of fair sequences

(Weak) fairness of actions: each action continuously enabled along a computation must eventually proceed.

Theorem (fair traces)

An infinite trace $\alpha_0\alpha_1\alpha_2\ldots$ is fair iff there exists a non-Zeno timed execution sequence

$$P_0 \xrightarrow{1}\xrightarrow{v_0} P1 \xrightarrow{1}\xrightarrow{v_1} \ldots P_n \xrightarrow{1}\xrightarrow{v_n} P_{n+1}\ldots$$

where $v_0 v_1 \ldots v_m \ldots = \alpha_0\alpha_1\ldots\alpha_i\ldots$

F. Corradini, M.R. Di Berardini, W. Vogler

Fairness of Actions in System Computations
Acta Informatica 43, pp. 73 130, 2006.

F. Corradini, M.R. Di Berardini, W. Vogler

Time and Fairness in a Process Algebra with Non-Blocking Reading
In Proc. of 35th Conference on Current Trends in Theory and Practice of Computer Science (SOFSEM 2009), LNCS 5404: 193 204 (2009).

. Costa, C. Stirling

Weak and Strong Fairness in CCS
Information and Computation 73, pp. 207-244, 1987

Liveness in mutual exclusion algorithms

Liveness in mutual exclusion algorithms

If, whenever at any computation a process P_i requests the execution of its critical section (e.g. req_i action), then in any continuation of that computation P_i will perform its critical section (e.g. cs_i action).

Liveness in mutual exclusion algorithms

Liveness in mutual exclusion algorithms

If, whenever at any computation a process P_i requests the execution of its critical section (e.g. req_i action), then in any continuation of that computation P_i will perform its critical section (e.g. cs_i action).

- To establish that an algorithm preserves its liveness property, we check if any occurrence of req_i in a fair trace is eventually followed by cs_i.
- Namely, we check if the process is free from catastrophic cycles.

F. Corradini and W. Vogler

Measuring the performance of asynchronous systems with PAFAS
Theor. Comput. Sci 335(2-3): 187-213 (2005).

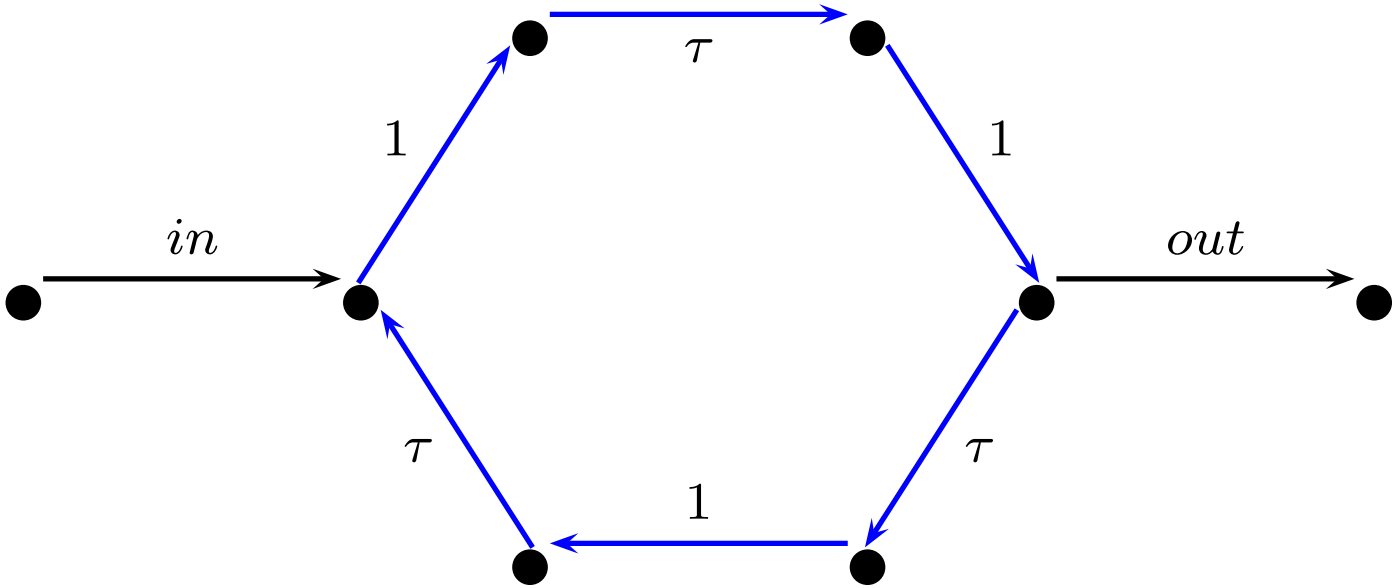
F. Corradini, M.R. Di Berardini, W. Vogler

Time and Fairness in a Process Algebra with Non-Blocking Reading
In Proc. of 35th Conference on Current Trends in Theory and Practice of Computer Science (SOFSEM 2009), LNCS 5404: 193 204 (2009).

Catastrphic cycles

Catastrphic cycles

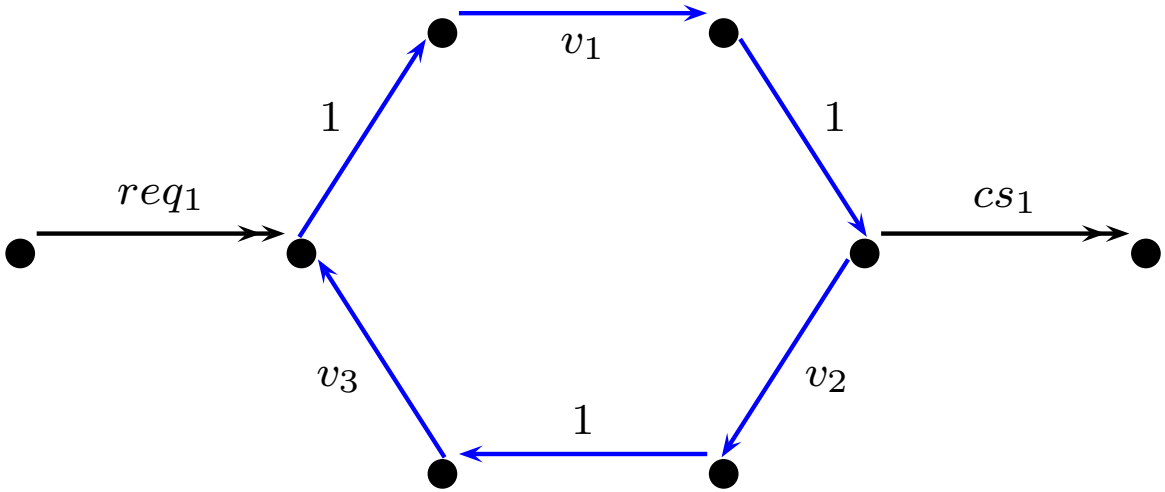
A cycle in the rRTS(P) is called catastrophic if it contains a positive number of time steps but no in's and no out's.



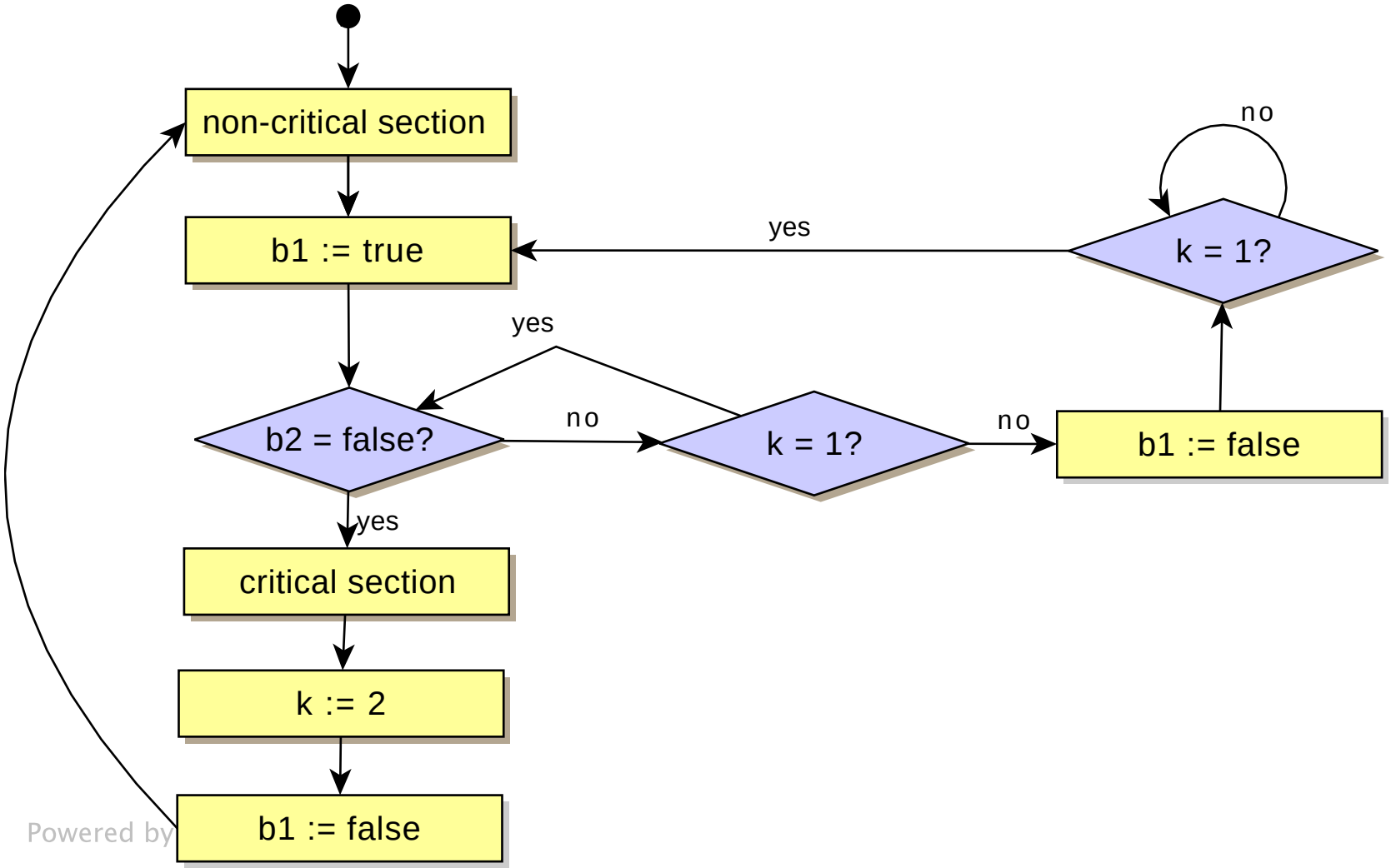
Catastrphic cycles and live traces

Catastrphic cycles and liveness

In a fair trace, after the request req_i there exists an infinite sequence of actions and time steps without any cs_i action.



Dekker algorithm



Dekker algorithm translated in PAFAS

```
B1F = b1rf.B1F + b1wf.B1F + b1wt.B1T;
B1T = b1rt.B1T + b1wt.B1T + b1wf.B1F;
B2F = b2rf.B2F + b2wf.B2F + b2wt.B2T;
B2T = b2rt.B2T + b2wt.B2T + b2wf.B2F;

K1 = kr1.K1 + kw1.K1 + kw2.K2;
K2 = kr2.K2 + kw2.K2 + kw1.K1;
PV = (B1F |[]| B2F |[]| K1);

P1 = in.b1wt.P11;
P11 = b2rf.P14 + b2rt.P12;
P12 = kr1.P11 + kr2.b1wf.P13;
P13 = kr1.b1wt.P11 + kr2.P13;
P14 = out.kw2.b1wf.P1;

P2 = req2.b2wt.P21;
P21 = b1rf.P24 + b1rt.P22;
P22 = kr2.P21 + kr1.b2wf.P23;
P23 = kr2.b2wt.P21 + kr1.P23;
P24 = exit2.kw1.b2wf.P2;

DEKKER = ((P1 |[]| P2)|[B]|PV)[L -> tau];
```

$B = sort(PV)$, $L = sort(DEKKER) \backslash \{in, out\}$

Liveness in Dekker - some results

Result on Dekker

Dekker **is not live** meaning that catastrophic cycles are detected.

Liveness in Dekker - some results

Result on Dekker

Dekker **is not live** meaning that catastrophic cycles are detected.

- Some unwanted behaviour: **a process reading/writing a variable can indefinitely block another process that tries to read/write it.**

Liveness in Dekker - some results

Result on Dekker

Dekker **is not live** meaning that catastrophic cycles are detected.

- Some unwanted behaviour: **a process reading/writing a variable can indefinitely block another process that tries to read/write it.**
- Dekker is not live under the assumption of fairness of actions.
- Suffers of Starvation and livelock.

F. Corradini, M.R. Di Berardini, W. Vogler

Checking a Mutex Algorithm in a Process Algebra with Fairness

Proc. of CONCUR'06, pp. 142-157, LNCS 4137, 2006

Liveness in Dekker - some results

- We could consider the scenario where multiple concurrent processes can read the same variable:

$$B_i(false) = \{b_i rf\} > b_i wf.B_i(false) + b_i wt.B_i(true)$$
$$B_i(true) = \{b_i rt\} > b_i wt.B_i(true) + b_i wf.B_i(false)$$

Liveness in Dekker - some results

- We could consider the scenario where multiple concurrent processes can read the same variable:

$$B_i(false) = \{b_i rf\} > b_i wf.B_i(false) + b_i wt.B_i(true)$$
$$B_i(true) = \{b_i rt\} > b_i wt.B_i(true) + b_i wf.B_i(false)$$

- Dekker is still not live.
- Catastrophic cycles are detected.
- Livelock is still present.

Liveness in Dekker - further considerations

- The only ordinary actions are those actions that correspond to the writing of a new value
- These actions can be tought of as non-destructive operations, allowing other potential concurrent accesses
- This way of accessing variables is not new, e.g database systems.

$$B_i(false) = \{b_i rf, b_i wf\} > b_i wt.B_i(true)$$
$$B_i(true) = \{b_i rt, b_i wt\} > b_i wf.B_i(false)$$

- Dekker is live and no catastrophic cycle is detected.

F. Corradini, M.R. Di Berardini, W. Vogler

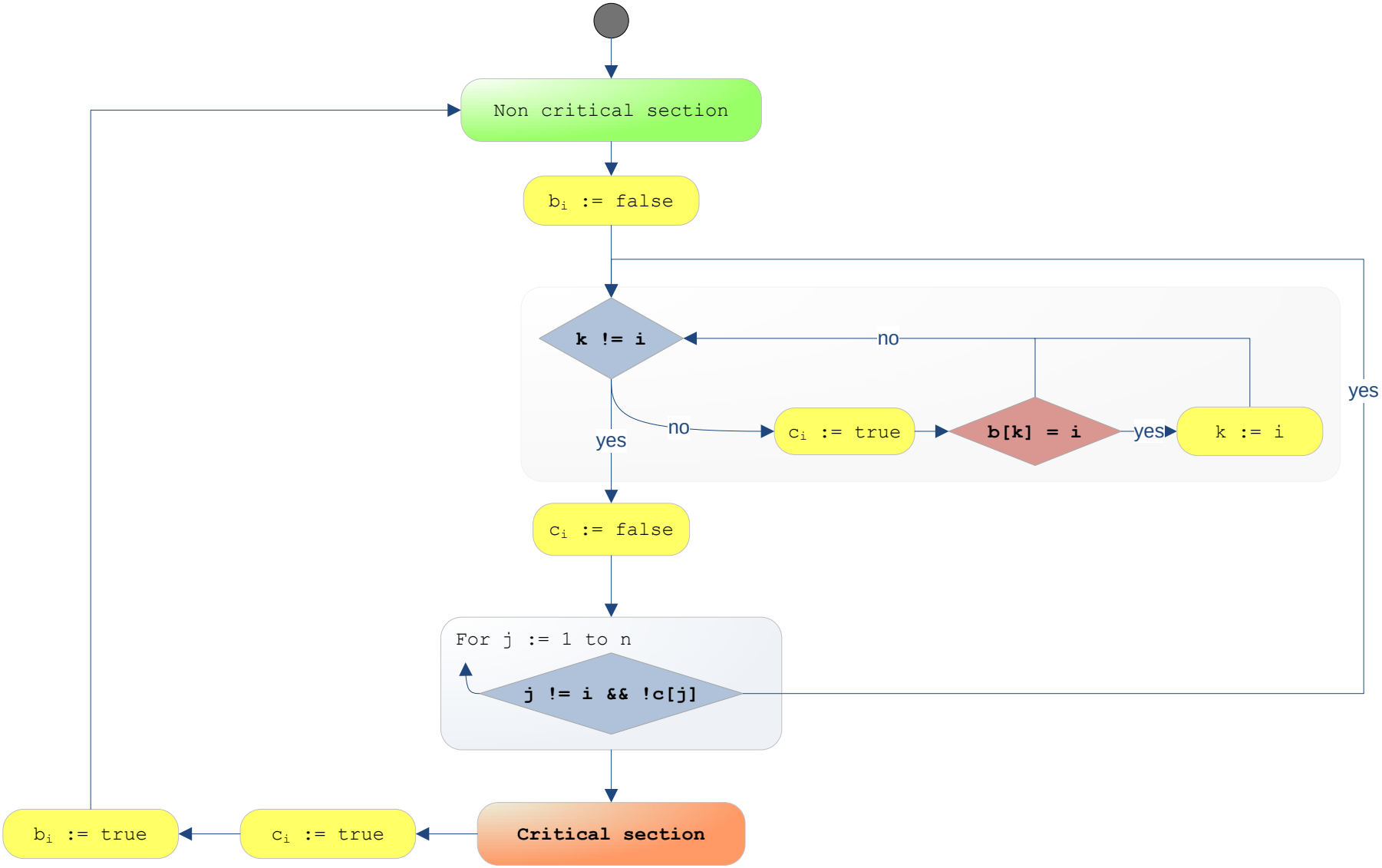
Time and Fairness in a Process Algebra with Non-Blocking Reading
In Proc. of 35th Conference on Current Trends in Theory and Practice of Computer Science (SOFSEM 2009), LNCS 5404: 193 204 (2009).

Dekker's algorithm

```
B1F = {b1rf, b1wf} > b1wt.B1T;
B1T = {b1rt, b1wt} > b1wf.B1F;
B2F = {b2rf, b2wf} > b2wt.B2T;
B2T = {b2rt, b2wt} > b2wf.B2F;

K1 = {kr1, kw1} > kw2.K2;
K2 = {kr2, kw2} > kw1.K1;

PV = (B1F |[]| B2F |[]| K1);

P1 = req1.b1wt.P11;
P11 = b2rf.P14 + b2rt.P12;
P12 = kr1.P11 + kr2.b1wf.P13;
P13 = kr1.b1wt.P11 + kr2.P13;
P14 = exit1.kw2.b1wf.P1;

P2 = req2.b2wt.P21;
P21 = b1rf.P24 + b1rt.P22;
P22 = kr2.P21 + kr1.b2wf.P23;
P23 = kr2.b2wt.P21 + kr1.P23;
P24 = exit2.kw1.b2wf.P2;

DEKKER = ((P1 |[]| P2)|[B]|PV)[L -> tau];
```

$B = sort(PV)$, $L = sort(DEKKER) \backslash \{in, out\}$

Dijkstra's algorithm

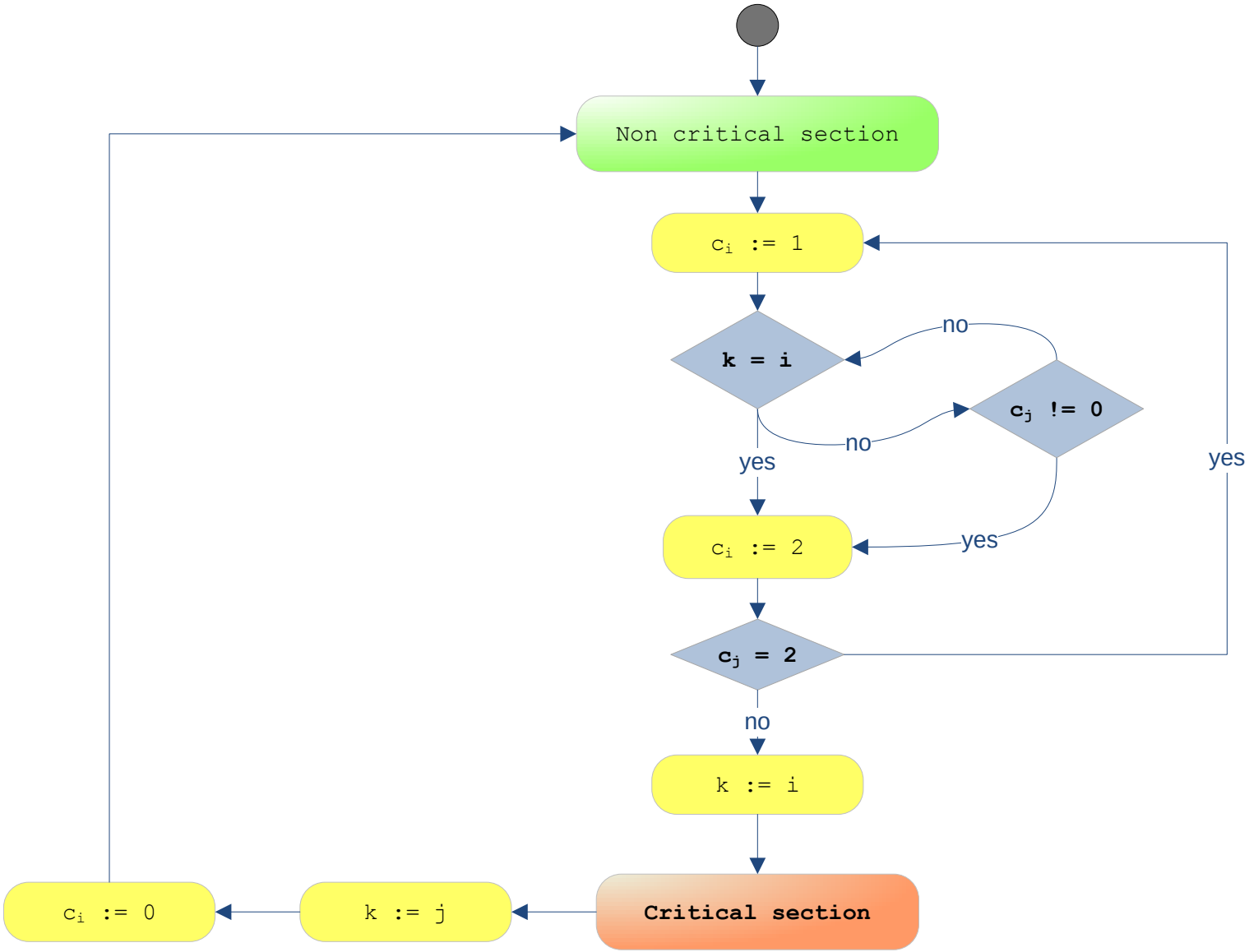


Dijkstra's algorithm

```
B1F = b1rf.B1F + b1wf.B1F + b1wt.B1T;          B2F = b2rf.B2F + b2wf.B2F + b2wt.B2T;
B1T = b1rt.B1T + b1wt.B1T + b1wf.B1F;          B2T = b2rt.B2T + b2wt.B2T + b2wf.B2F;
C1F = c1rf.C1F + c1wf.C1F + c1wt.C1T;          C2F = c2rf.C2F + c2wf.C2F + c2wt.C2T;
C1T = c1rt.C1T + c1wt.C1T + c1wf.C1F;          C2T = c2rt.C2T + c2wt.C2T + c2wf.C2F;

   K1 = kr1.K1 + kw1.K1 + kw2.K2 + get.(k1r1.put.K1 + k2r1.put.K1);
   K2 = kr2.K2 + kw2.K2 + kw1.K1 + get.(k1r2.put.K2 + k2r2.put.K2);

   PV = (B1T || B2T || C1T || C2T || K1);

P1 = b1wf.in.P11;                              P2 = b2wf.req2.P21;
P11 = kr1.P13 + kr2.c1wt.P12;                  P21 = kr2.P23 + kr1.c2wt.P22;
P12 = get.(k1r1.(b1rt.put.kw1.P11 +            P22 = get.(k2r2.(b2rt.put.kw2.P21 +
b1rf.put.P11) + k1r2.(b2rt.put.kw1.P11 +       b2rf.put.P21) + k2r1.(b1rt.put.kw2.P21 +
b2rf.put.P11));                                b1rf.put.P21));
P13 = c1wf.(c2rf.P11 + c2rt.P14); P14 =        P23 = c2wf.(c1rf.P21 + c1rt.P24);
out.c1wt.b1wt.P1;                              P24 = cs2.c2wt.b2wt.P2;

   DIJKSTRA = ((P1 |[]| P2) |[B]| PV)[L -> tau];
```

$B = sort(PV)$, $L = sort(DIJKSTRA) \backslash \{in, out\}$

Knuth's algorithm



Knuth's algorithm

```
C10 = c1w0.C10 + c1w1.C11 + c1w2.C12 +
c1r0.C10;
C11 = c1w0.C10 + c1w1.C11 + c1w2.C12 +
c1r1.C11;
C12 = c1w0.C10 + c1w1.C11 + c1w2.C12 +
c1r2.C12;
```

```
C20 = c2w0.C20 + c2w1.C21 + c2w2.C22 +
c2r0.C20;
C21 = c2w0.C20 + c2w1.C21 + c2w2.C22 +
c2r1.C21;
C22 = c2w0.C20 + c2w1.C21 + c2w2.C22 +
c2r2.C22;
```

```
K1 = kr1.K1 + kw1.K1 + kw2.K2;
K2 = kr2.K2 + kw2.K2 + kw1.K1
```

```
PV = (C10 || C20 || K1);
```

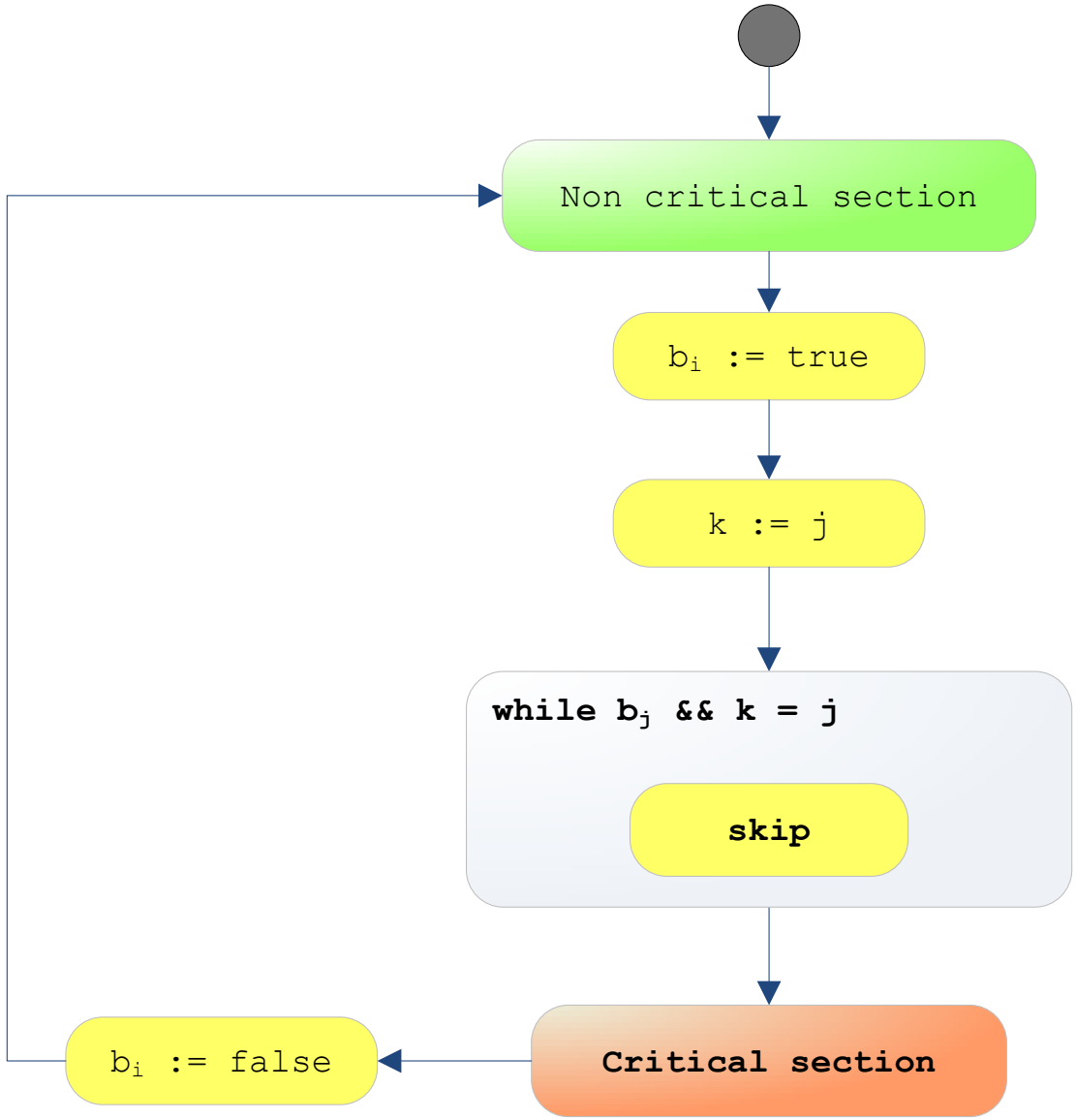
```
P1 = c1w1.in.P11;
P11 = kr1.P13 + kr2.P12;
P12 = c2r0.P13 + c2r1.P11 + c2r2.P11;
P13 = c1w2.P14;
P14 = c2r0.P15 + c2r1.P15 + c2r2.P16;
P15 = kw1.out.kw2.c1w0.P1;
P16 = c1w1.P11;
```

```
P2 = c2w1.req2.P21;
P21 = kr2.P23 + kr1.P22;
P22 = c1r0.P23 + c1r1.P21 + c1r2.P21;
P23 = c2w2.P24;
P24 = c1r0.P25 + c1r1.P25 + c1r2.P26;
P25 = kw2.cs2.kw1.c2w0.P2;
P26 = c2w1.P21;
```

```
KNUTH = ((P1 |[]| P2) |[B]| PV)[L -> tau];
```

$B = sort(PV)$, $L = sort(KNUTH) \backslash \{in, out\}$

Peterson's algorithm



Peterson's algorithm

```
B1F = b1rf.B1F + b1wf.B1F + b1wt.B1T;
B1T = b1rt.B1T + b1wt.B1T + b1wf.B1F;
```

```
B2F = b2rf.B2F + b2wf.B2F + b2wt.B2T;
B2T = b2rt.B2T + b2wt.B2T + b2wf.B2F;
```

```
K1 = kr1.K1 + kw1.K1 + kw2.K2;
K2 = kr2.K2 + kw2.K2 + kw1.K1;
```

```
PV = (B1F || B2F || K1);
```

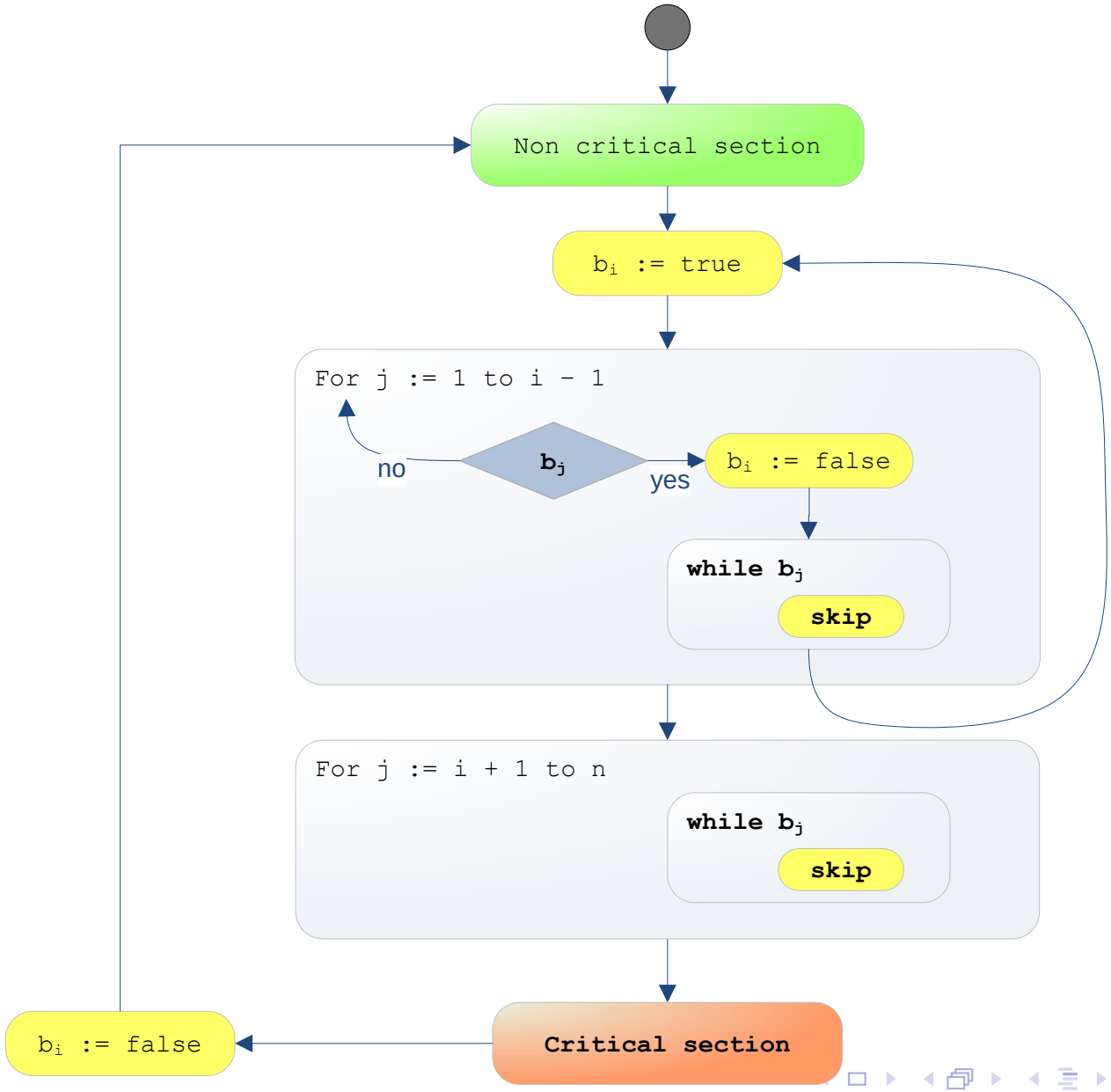
```
P1 = b1wt.in.kw2.P11;
P11 = b2rf.P12 + b2rt.(kr2.P11 +
kr1.P12);
P12 = out.b1wf.P1;
```

```
P2 = b2wt.req2.kw1.P21;
P21 = b1rf.P22 + b1rt.(kr1.P21 +
kr2.P22);
P22 = cs2.b2wf.P2;
```

```
PETERSON = ((P1 |[]| P2) |[B]| PV)[L -> tau];
```

$B = sort(PV)$, $L = sort(PETERSON) \backslash \{in, out\}$

Lamport's algorithm



Lamport's algorithm

```
B1F =b1rf.B1F +b1wf.B1F + b1wt.B1T;
B1T=b1rt.B1T +b1wf.B1F + b1wt.B1T;
```

```
B2F =b2rf.B2F +b2wf.B2F + b2wt.B2T;
B2T=b2rt.B2T +b2wf.B2F + b2wt.B2T;
```

```
PV = (B1F||B2F);
```

```
P1 = b1wt.in.P11;
P11 = b2rf.P12+b2rt.P11;
P12 = out.b1wf.P1;
```

```
P2 = b2wt.req2.P21;
P21 = b1rf.P23 + b1rt.b2wf.P22;
P22 = b1rf.b2wt.P21 + b1rt.P22;
P23 = cs2.b2wf.P2;
```

```
LAMPORT = ((P1 |[]| P2) |[B]| PV)[L -> tau];
```

$B = sort(PV)$, $L = sort(LAMPORT) \backslash \{in, out\}$

Relating results

- All the algorithms have been coded in PAFAS and automatically checked by the tool `FASE`.
- Interesting results have been found:

Algorithm	*Walker*	*Without read acts*	*With read acts*
Dekker	*not live*	*not live*	*live*
Dijkstra	*not live*	*not live*	*not live*
Knuth	*live*	*not live*	*not live*
Peterson	*live*	*not live*	*live*
$Lamport_{live_{p_1} \wedge live_{P_2}}$	*not live*	*not live*	*not live*
$Lamport_{live_{p_1} \wedge \neg live_{P_2}}$	*live*	*not live*	*live*

Table: Comparing Walker result with catastrophic cycles detection

Further observations

Peterson's algorithm

- Without using read actions, FASE has detected catastrophic cycles: **P_i continuously reads the same value from the same variable while P_2 is stuck and waits to write it.**
- When read acts are used to models the action of read the same value from a variable, Peterson's algorithm becomes live.

Further observations

Peterson's algorithm

- Without using read actions, `FASE` has detected catastrophic cycles: **P_i continuously reads the same value from the same variable while P_2 is stuck and waits to write it.**
- When read acts are used to models the action of read the same value from a variable, Peterson's algorithm becomes live.

Lamport's algorithm

- The same considerations about Peterson's algorithms are valid if we consider to observe P_1.
- If we observe P_2, than the catastrophic cycles are detected in both cases as expected (due to the asymmetry of the processes).

Further observations

Dijkstra's algorithm

- In Dijkstra's algorithm, `FASE` has detected catastrophic cycles in both cases.
- It is known that this algorithm is susceptible to starvation.
- But there are also catastrophic cycles as following:

Further observations

Dijkstra's algorithm

- In Dijkstra's algorithm, `FASE` has detected catastrophic cycles in both cases.
- It is known that this algorithm is susceptible to starvation.
- But there are also catastrophic cycles as following:

$$\underline{K2} = \{\underline{kr2}, kw2\} > (kw1.K1 + \underline{get}.(k1r2.put.K2 + k2r2.put.K2));$$

$$\underline{P12} = \underline{get}.(\ldots);$$

$$\underline{P21} = \underline{kr2}.P23 + kr1.c2wt.P22;$$

$$P = \underline{P_1 2} \parallel \underline{P_2 2} \parallel_B \ldots K2 \xrightarrow{\tau(get)} \xrightarrow{\tau} \ldots \xrightarrow{1} P$$

Further observations

Knuth's algorithm

- FASE has detected catastrophic cycles in both cases.
- When read acts are used, the follow catastrophic cycles is detected:

$$C12 = \{c1r2, c1w2\} > (c1w0.C10 + c1w1.C11);$$

$$C21 = \{c2r1, c2w1\} > (c2w0.C20 + c2w2.C22);$$
$$C22 = \{c2r2, c2w2\} > (c2w0.C20 + c2w1.C21);$$

$$P14 = c2r0.P15 + c2r1.P15 + c2r2.P16;$$

$$P21 = kr2.P23 + kr1.P22$$
$$P22 = c1r0.P23 + c1r1.P21 + c1r2.P21$$
$$P23 = c2w2.P24$$
$$P24 = c1r0.P25 + c1r1.P25 + c1r2.P26$$
$$P26 = c2w1.P21$$

Further observations

Knuth's algorithm

- FASE has detected catastrophic cycles in both cases.
- When read acts are used, the follow catastrophic cycles is detected:

$$C12 = \{c1r2, c1w2\} > (c1w0.C10 + c1w1.C11);$$

$$C21 = \{c2r1, c2w1\} > (c2w0.C20 + c2w2.C22);$$
$$C22 = \{c2r2, c2w2\} > (c2w0.C20 + c2w1.C21);$$

$$P14 = c2r0.P15 + c2r1.P15 + c2r2.P16;$$

$$P21 = kr2.P23 + kr1.P22$$
$$P22 = c1r0.P23 + c1r1.P21 + c1r2.P21$$
$$P23 = c2w2.P24$$
$$P24 = c1r0.P25 + c1r1.P25 + c1r2.P26$$
$$P26 = c2w1.P21$$

$$P = P14 \parallel P24 \parallel_B (C12 \parallel C22 \parallel K(2)) \xrightarrow{1}$$

Further observations

Knuth's algorithm

- FASE has detected catastrophic cycles in both cases.
- When read acts are used, the follow catastrophic cycles is detected:

$$C12 = \{c1r2, c1w2\} > (c1w0.C10 + c1w1.C11);$$

$$C21 = \{c2r1, c2w1\} > (c2w0.C20 + c2w2.C22);$$
$$C22 = \{c2r2, c2w2\} > (c2w0.C20 + c2w1.C21);$$

$$P14 = c2r0.P15 + c2r1.P15 + c2r2.P16;$$

$$P21 = kr2.P23 + kr1.P22$$
$$P22 = c1r0.P23 + c1r1.P21 + c1r2.P21$$
$$P23 = c2w2.P24$$
$$P24 = c1r0.P25 + c1r1.P25 + c1r2.P26$$
$$P26 = c2w1.P21$$

$$P = P14 \parallel P24 \parallel_B (C12 \parallel C22 \parallel K(2)) \xrightarrow{1}$$
$$P' = \underline{P14} \parallel \underline{P24} \parallel_B (\underline{C12} \parallel \underline{C22} \parallel K(2)) \xrightarrow{\tau(c2w1)}$$

Further observations

Knuth's algorithm

- FASE has detected catastrophic cycles in both cases.
- When read acts are used, the follow catastrophic cycles is detected:

$$C12 = \{c1r2, c1w2\} > (c1w0.C10 + c1w1.C11);$$

$$C21 = \{c2r1, c2w1\} > (c2w0.C20 + c2w2.C22);$$
$$C22 = \{c2r2, c2w2\} > (c2w0.C20 + c2w1.C21);$$

$$P14 = c2r0.P15 + c2r1.P15 + c2r2.P16;$$

$$P21 = kr2.P23 + kr1.P22$$
$$P22 = c1r0.P23 + c1r1.P21 + c1r2.P21$$
$$P23 = c2w2.P24$$
$$P24 = c1r0.P25 + c1r1.P25 + c1r2.P26$$
$$P26 = c2w1.P21$$

$$P = P14 \parallel P24 \parallel_B (C12 \parallel C22 \parallel K(2)) \xrightarrow{1}$$
$$P' = \underline{P14} \parallel \underline{P24} \parallel_B (\underline{C12} \parallel \underline{C22} \parallel K(2)) \xrightarrow{\tau(c2w1)}$$
$$P'' = P14 \parallel P21 \parallel_B (C12 \parallel C21 \parallel K(2))$$

Further observations

Knuth's algorithm

- FASE has detected catastrophic cycles in both cases.
- When read acts are used, the follow catastrophic cycles is detected:

$$C12 = \{c1r2, c1w2\} > (c1w0.C10 + c1w1.C11);$$

$$C21 = \{c2r1, c2w1\} > (c2w0.C20 + c2w2.C22);$$
$$C22 = \{c2r2, c2w2\} > (c2w0.C20 + c2w1.C21);$$

$$P14 = c2r0.P15 + c2r1.P15 + c2r2.P16;$$

$$P21 = kr2.P23 + kr1.P22$$
$$P22 = c1r0.P23 + c1r1.P21 + c1r2.P21$$
$$P23 = c2w2.P24$$
$$P24 = c1r0.P25 + c1r1.P25 + c1r2.P26$$
$$P26 = c2w1.P21$$

$$P = P14 \parallel P24 \parallel_B (C12 \parallel C22 \parallel K(2)) \xrightarrow{1}$$
$$P' = \underline{P14} \parallel \underline{P24} \parallel_B (\underline{C12} \parallel \underline{C22} \parallel K(2)) \xrightarrow{\tau(c2w1)}$$
$$P'' = P14 \parallel P21 \parallel_B (C12 \parallel C21 \parallel K(2)) \xrightarrow{1}$$

Further observations

Knuth's algorithm

- FASE has detected catastrophic cycles in both cases.
- When read acts are used, the follow catastrophic cycles is detected:

$$C12 = \{c1r2, c1w2\} > (c1w0.C10 + c1w1.C11);$$

$$C21 = \{c2r1, c2w1\} > (c2w0.C20 + c2w2.C22);$$
$$C22 = \{c2r2, c2w2\} > (c2w0.C20 + c2w1.C21);$$

$$P14 = c2r0.P15 + c2r1.P15 + c2r2.P16;$$

$$P21 = kr2.P23 + kr1.P22$$
$$P22 = c1r0.P23 + c1r1.P21 + c1r2.P21$$
$$P23 = c2w2.P24$$
$$P24 = c1r0.P25 + c1r1.P25 + c1r2.P26$$
$$P26 = c2w1.P21$$

$$P = P14 \parallel P24 \parallel_B (C12 \parallel C22 \parallel K(2)) \xrightarrow{1}$$
$$P' = \underline{P14} \parallel \underline{P24} \parallel_B (\underline{C12} \parallel \underline{C22} \parallel K(2)) \xrightarrow{\tau(c2w1)}$$
$$P'' = P14 \parallel P21 \parallel_B (C12 \parallel C21 \parallel K(2)) \xrightarrow{1} \xrightarrow{\tau(kr2)}$$

Further observations

Knuth's algorithm

- FASE has detected catastrophic cycles in both cases.
- When read acts are used, the follow catastrophic cycles is detected:

$$C12 = \{c1r2, c1w2\} > (c1w0.C10 + c1w1.C11);$$

$$C21 = \{c2r1, c2w1\} > (c2w0.C20 + c2w2.C22);$$
$$C22 = \{c2r2, c2w2\} > (c2w0.C20 + c2w1.C21);$$

$$P14 = c2r0.P15 + c2r1.P15 + c2r2.P16;$$

$$P21 = kr2.P23 + kr1.P22$$
$$P22 = c1r0.P23 + c1r1.P21 + c1r2.P21$$
$$P23 = c2w2.P24$$
$$P24 = c1r0.P25 + c1r1.P25 + c1r2.P26$$
$$P26 = c2w1.P21$$

$$P = P14 \parallel P24 \parallel_B (C12 \parallel C22 \parallel K(2)) \xrightarrow{1}$$
$$P' = \underline{P14} \parallel \underline{P24} \parallel_B (\underline{C12} \parallel \underline{C22} \parallel K(2)) \xrightarrow{\tau(c2w1)}$$
$$P'' = P14 \parallel P21 \parallel_B (C12 \parallel C21 \parallel K(2)) \xrightarrow{1} \xrightarrow{\tau(kr2)} \xrightarrow{\tau(c2w2)} P$$

And...

Thanks!