

# Time and Fairness in a Process Algebra with Non-Blocking Readings

F. Corradini<sup>1</sup>   M.R. Di Berardini<sup>1</sup>   W. Vogler<sup>2</sup>

<sup>1</sup>Dip. di Matematica e Informatica  
Università di Camerino

<sup>2</sup>Institut für Informatik  
Universität Augsburg

January 26, 2009



# Modelling non-destructive behaviours

- In Petri Nets, **non-destructive reading operations** are modelled with a special kind of arcs called **read arcs** (but also condition/test arcs)
- Read arcs represent **positive context conditions**, i.e elements needed for an event to occur, but not effected by it

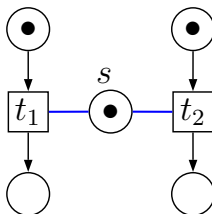


Figure: Transitions  $t_1$  and  $t_2$  can fire concurrently

# Modelling non-destructive behaviours

In ordinary Petri Nets, non-destructive operations have to be rendered as a consume/produce loop

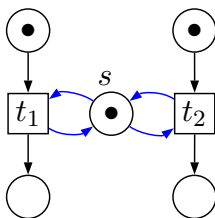


Figure: Transitions  $t_1$  and  $t_2$  cannot read the resource  $s$  concurrently

An explicit representation of positive contexts allows:

- 1 a faithful representation of systems where the notion of “reading without consuming” is commonly used (e.g. databases, concurrent constraint programming, etc.)
- 2 to specify directly and naturally a level of concurrency greater than in classical nets

Moreover, read arcs add relevant expressivity:



W. Vogler.

Efficiency of Asynchronous Systems, Read Arcs and the MUTEX-problem.

*Theoretical Computer Science* 275(1-2), pp. 589-631, 2002

- We study expressivity of non-blocking behaviours in the setting of a timed process algebra — called PAFAS
- We are mainly interested in the impact that non-blocking behaviours have on
  - timing,
  - fairness and
  - liveness of systems

- 1 Introduction
- 2 A Process Algebra with Non-Blocking Readings**
- 3 Fairness and Timing
- 4 Dekker's Algorithm and its liveness property

# A basic PA for describing read behaviours

$P ::=$	$\text{nil}$	
	$  \alpha.P$	<i>action-prefix</i> $\alpha$ is either visible ( $a, b, c, \dots$ ) or $\tau$
	$  P_1 + P_2$	<i>nondeterministic choice</i>
	$  P_1 \parallel_A P_2$	<i>CSP-like parallel composition</i> $A$ is a set of visible actions
	$  P[\Phi]$	$\Phi$ is a relabeling function
	$  \text{rec } x.P$	<i>recursive behaviour</i>

# A basic PA for describing read behaviours

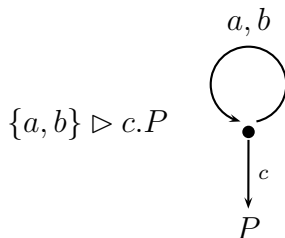
$P ::=$	$\text{nil}$	
	$  \alpha.P$	<i>action-prefix</i> $\alpha$ is either visible ( $a, b, c, \dots$ ) or $\tau$
	$  P_1 + P_2$	<i>nondeterministic choice</i>
	$  P_1 \parallel_A P_2$	<i>CSP-like parallel composition</i> $A$ is a set of visible actions
	$  P[\Phi]$	$\Phi$ is a relabeling function
	$  \text{rec } x.P$	<i>recursive behaviour</i>
	$  \{\alpha_1, \dots, \alpha_n\} \triangleright P$	<i>read-set-prefix operator</i>

A term like

$$\{\alpha_1, \dots, \alpha_n\} \triangleright P$$

models a variable (or a more complex data structure) that behaves as  $P$  but can also be read with actions in the read-set  $\{\alpha_1, \dots, \alpha_n\}$

EX:



# Functional Behaviour

$$\text{ACT} \frac{}{\alpha.P \xrightarrow{\alpha} P} \qquad \text{SUM} \frac{P_1 \xrightarrow{\alpha} P'}{P_1 + P_2 \xrightarrow{\alpha} P'} + \text{symm.}$$

$$\text{SYNCH} \frac{\alpha \in A, P_1 \xrightarrow{\alpha} P'_1, P_2 \xrightarrow{\alpha} P'_2}{P_1 \parallel_A P_2 \xrightarrow{\alpha} P'_1 \parallel_A P'_2}$$

$$\text{PAR} \frac{\alpha \notin A, P_1 \xrightarrow{\alpha} P'_1}{P_1 \parallel_A P_2 \xrightarrow{\alpha} P'_1 \parallel_A P_2} + \text{symm.}$$

# Functional Behaviour

$$\text{ACT} \frac{}{\alpha.P \xrightarrow{\alpha} P} \qquad \text{SUM} \frac{P_1 \xrightarrow{\alpha} P'}{P_1 + P_2 \xrightarrow{\alpha} P'} + \text{symm.}$$

$$\text{SYNCH} \frac{\alpha \in A, P_1 \xrightarrow{\alpha} P'_1, P_2 \xrightarrow{\alpha} P'_2}{P_1 \parallel_A P_2 \xrightarrow{\alpha} P'_1 \parallel_A P'_2}$$

$$\text{PAR} \frac{\alpha \notin A, P_1 \xrightarrow{\alpha} P'_1}{P_1 \parallel_A P_2 \xrightarrow{\alpha} P'_1 \parallel_A P_2} + \text{symm.}$$

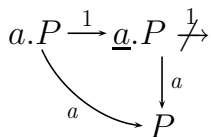
$$\text{READ}_1 \frac{\alpha \in \{\alpha_1, \dots, \alpha_n\}}{\{\alpha_1, \dots, \alpha_n\} \triangleright P \xrightarrow{\alpha} \{\alpha_1, \dots, \alpha_n\} \triangleright P}$$

$$\text{READ}_2 \frac{P \xrightarrow{\alpha} P'}{\{\alpha_1, \dots, \alpha_n\} \triangleright P \xrightarrow{\alpha} P'}$$

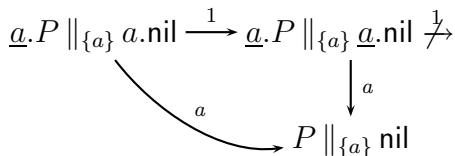
## Basic Assumption

Actions have an **upper time bound** – 0 or 1 – as a maximal delay, where:

- $\alpha.P$  and  $\{\alpha_1, \dots, \alpha_n\} \triangleright P$  denote **patient prefixes** – time bound 1
- $\underline{\alpha}.P$  and  $\{\underline{\alpha}_1, \dots, \underline{\alpha}_n\} \triangleright Q$  denote **urgent prefixes** – time bound 0



as stand-alone process  $\underline{a}.P$  but  
has no reason to wait



as a component of a larger system, it can  
wait for a synchronization

**Initial processes** ranged over by  $P, P', \dots$

**General processes** ranged over by  $Q, Q', \dots$

Two transition relations:

- 1 Functional Behaviour:  $Q \xrightarrow{\alpha} Q'$
- 2 Temporal Behaviour:  $Q \xrightarrow{1} Q'$

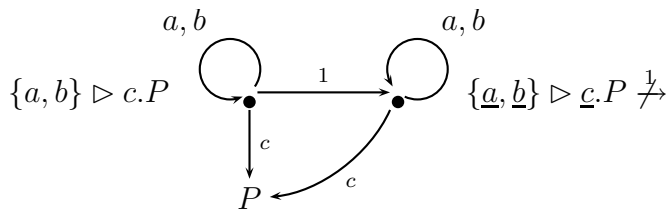
Whenever

$$Q \xrightarrow{1} Q'$$

all actions that are enabled in  $Q$  become urgent in  $Q'$ , but ...

- $Q \xrightarrow{1}$  only if it has no urgent actions
- $a$  is **urgent** in  $Q' = \underline{a}.P \parallel_{\{a\}} \underline{a}.nil$ , but not in  $Q = \underline{a}.P \parallel_{\{a\}} a.nil$

# Urgent Non-Blocking Actions



- 1 Introduction
- 2 A Process Algebra with Non-Blocking Readings
- 3 Fairness and Timing**
- 4 Dekker's Algorithm and its liveness property

- **Weak Fairness of Actions** requires that an action continuously enabled along a computation must eventually proceed
- We have proven that
  - fair traces of untimed processes (as defined by Costa and Stirling)
  - can be characterized in terms of
  - everlasting (non-Zeno) timed execution sequences



F. Corradini, M.R. Di Berardini, W. Vogler  
Fairness of Actions in System Computations  
*Acta Informatica* **43**, pp. 73-130, 2006



G. Costa, C. Stirling  
Weak and Strong Fairness in CCS  
*Information and Computation* **73**, pp. 207-244, 1987

# A characterization of fair sequences

## Theorem (fair traces — the infinite case)

Let  $P_0 \in \mathbb{P}_1$  and  $\alpha_0, \alpha_1, \alpha_2, \dots \in \mathbb{A}_\tau$ . An infinite trace  $\alpha_0 \alpha_1 \alpha_2 \dots$  of  $P_0$  is **fair** iff there exists a non-Zeno timed execution sequence

$$P_0 \xrightarrow{1} \xrightarrow{v_0} P_1 \xrightarrow{1} \xrightarrow{v_1} P_2 \dots P_n \xrightarrow{1} \xrightarrow{v_n} P_{n+1} \dots$$

where  $v_0 v_1 \dots v_m \dots = \alpha_0 \alpha_1 \dots \alpha_j \dots$

Each step of the form

$$P_i \xrightarrow{1} \xrightarrow{v_i} P_{i+1}$$

is a **locally fair step**



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

Time and Fairness in a Process Algebra with Non-Blocking Reading

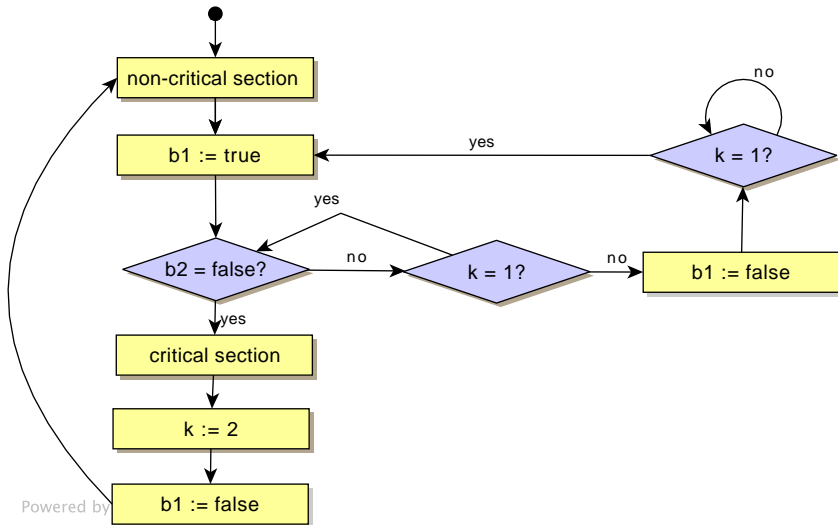
TR 2008-13, Institute of Computer Science, University of Augsburg, 2008

- 1 Introduction
- 2 A Process Algebra with Non-Blocking Readings
- 3 Fairness and Timing
- 4 Dekker's Algorithm and its liveness property**

# Dekker's Algorithm

- It is one of the most classic algorithm for the MUTEX-problem
- The entry protocol is based on the values of two **request variables** (or status flag) and a **turn variable**
- The request variables  $b_1$  and  $b_2$  take values in  $\{true, false\}$ 
  - initially, both  $b_1$  and  $b_2$  are set to *false*
  - $b_i$  is *true* if  $P_i$  is requesting to entry its critical section
  - only  $P_i$  writes  $b_i$ , but both can read it
- The turn variable  $k$  takes values in  $\{1, 2\}$ 
  - initially  $k$  is set to 1
  - $k$  is  $i$  if it is the  $P_i$ 's turn to enter the critical section
  - both  $P_1$  and  $P_2$  can read and write  $k$

# Dekker's Algorithm



Powered by

# Liveness Property of Dekker's Algorithm

- Dekker's algorithm and its properties (namely, mutual exclusion, and **liveness**) have been studied by Walker
- Liveness requires that: *whenever, at some point, a process  $P_i$  requests the execution of its critical section, then at some later point  $P_i$  will enter it*
- The verification of liveness properties usually requires some fairness assumption. Is weak fairness able to ensure this property?
- It depends on our ability to prevent unwanted behaviours like : *a process reading a variable can **indefinitely** blocks another process trying to write it*



D.J. Walker

Automated Analysis of Mutual Exclusion algorithms using CCS

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

# A standard representation of program variables

- Let us consider:

a program variable  $V = r.V + w.V$   
a reading activity  $R = r.R$   
a writing activity  $W = w.W$   
and  $P = (R \parallel_{\emptyset} W) \parallel_{\{r,w\}} V$

- According to our timed operational semantics

$$\begin{aligned} P &\xrightarrow{1} (\underline{r}.R \parallel_{\emptyset} \underline{w}.W) \parallel_{\{r,w\}} (\underline{r}.V + \underline{w}.V) \\ &\xrightarrow{r} (R \parallel_{\emptyset} \underline{w}.W) \parallel_{\{r,w\}} V \approx P \end{aligned}$$

- Since

$$P \xrightarrow{1} \xrightarrow{r} P \xrightarrow{1} \xrightarrow{r} P \dots$$

$P \xrightarrow{r} \xrightarrow{r} \dots$  is fair

# Dekker's program variables — version 1

If the processes representing program variables are defined as follows (all actions are “ordinary”)

$$B_i(\text{false}) = b_{i\text{rf}}.B_i(\text{false}) + (b_{i\text{wf}}.B_i(\text{false}) + b_{i\text{wt}}.B_i(\text{true}))$$

$$B_i(\text{true}) = b_{i\text{rt}}.B_i(\text{true}) + (b_{i\text{wf}}.B_i(\text{false}) + b_{i\text{wt}}.B_i(\text{true}))$$

$$K(1) = kr1.K(1) + (kw1.K(1) + kw2.K(2))$$

$$K(2) = kr2.K(1) + (kw1.K(1) + kw2.K(2))$$

Dekker's algorithm is **not live** also under the assumption of fairness of actions



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

Checking a Mutex Algorithm in a Process Algebra with Fairness

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

# A variable with non-blocking behaviours

- Let us consider:

a program variable  $V = \{r\} \triangleright w.V$   
a reading activity  $R = r.R$   
a writing activity  $W = w.W$   
and  $P = (R \parallel_{\emptyset} W) \parallel_{\{r,w\}} V$

- Now, a run from  $P$  consisting of infinitely many  $r$ 's is **not** fair

$$\begin{array}{l} P \xrightarrow{1} Q = (\underline{r}.R \parallel_{\emptyset} \underline{w}.W) \parallel_{\{r,w\}} \{\underline{r}\} \triangleright \underline{w}.V \\ \xrightarrow{r} Q' = (R \parallel_{\emptyset} \underline{w}.W) \parallel_{\{r,w\}} \{\underline{r}\} \triangleright \underline{w}.V \quad \xrightarrow{1} \\ \dots \\ \xrightarrow{r} Q' \\ \xrightarrow{w} P \end{array}$$

# A variable with non-blocking behaviours

- Now, readings cannot block writings ... but this is not enough yet
- Indeed:

$$P \begin{array}{l} \xrightarrow{1} \\ \xrightarrow{w} \end{array} (\underline{r}.R \parallel_{\emptyset} \underline{w}.W) \parallel_{\{r,w\}} \{ \underline{r} \} \triangleright \underline{w}.V \\ (\underline{r}.R \parallel_{\emptyset} W) \parallel_{\{r,w\}} V \approx P$$

- So, writings can still block readings
- We need something like  $V = \{r, w\} \triangleright \text{nil}$

## Dekker's program variables — version 2

- Let us consider a representation of program variables where the only ordinary actions are those corresponding to the writing of a new value

$$B_i(\text{false}) = \{b_{i\text{rf}}, b_{i\text{wf}}\} \triangleright b_{i\text{wt}}.B_i(\text{true}) \quad \text{for } i=1,2$$

$$B_i(\text{true}) = \{b_{i\text{rt}}, b_{i\text{wt}}\} \triangleright b_{i\text{wf}}.B_i(\text{false}) \quad \text{for } i=1,2$$

$$K(1) = \{kr1, kw1\} \triangleright kw2.K(2)$$

$$K(2) = \{kr2, kw2\} \triangleright kw1.K(1)$$

- Re-writing the same value does not change the state of the variable
- These actions can be thought of as non-destructive operations, allowing other potential concurrent accesses
- This way of accessing variables is not new, Ex: [The two-phase locking protocol](#)

# Some results about liveness of Dekker's

## Theorem

*Dekker is live*

## Theorem

*Dekker<sub>ℓ</sub> is **not** live*

*Dekker<sub>ℓ</sub>* is a variation of *Dekker* where:

$$B_i(\text{false}) = \{b_i rf\} \triangleright (b_i wf.B_i(\text{false}) + b_i wt.B_i(\text{true})) \quad \text{for } i=1,2$$

$$B_i(\text{true}) = \{b_i rt\} \triangleright (b_i wf.B_i(\text{false}) + b_i wt.B_i(\text{true})) \quad \text{for } i=1,2$$

$$K(1) = \{kr1\} \triangleright (kw1.K(1) + kw2.K(2))$$

$$K(2) = \{kr2\} \triangleright (kw1.K(1) + kw2.K(2))$$

- We have introduced the first process algebra with non-blocking reading actions for modelling concurrent asynchronous systems
- We have studied the impact this new kind of actions has on fairness, liveness and the timing of systems, using as application Dekker's mutual exclusion algorithm
- In particular, we have shown how non-blocking reading have a decisive impact on the liveness of Dekker's algorithm

Thank you for your attention

Any questions?

① Functional Behaviour:  $Q \xrightarrow{\alpha} Q'$

describes how  $Q$  evolves into  $Q'$  by performing the action  $\alpha$

② Refusal Behaviour:  $Q \xrightarrow{X}_r Q'$

describes how  $Q$  evolves into  $Q'$  by letting time pass

- it is a **conditional** time step of duration 1
- $X \subseteq \mathbb{A}$  containing actions that are not urgent in  $Q$

EX:  $a.P \xrightarrow{X}_r \underline{a}.P$  for each  $X \subseteq \mathbb{A}$ , but  $\underline{a}.P \xrightarrow{X}_r \underline{a}.P$  iff  $a \notin X$

# Transitional Semantics of PAFAS<sub>s</sub>

① Functional Behaviour:  $Q \xrightarrow{\alpha} Q'$

describes how  $Q$  evolves into  $Q'$  by performing the action  $\alpha$

② Refusal Behaviour:  $Q \xrightarrow{X}_r Q'$

describes how  $Q$  evolves into  $Q'$  by letting time pass

- it is a **conditional** time step of duration 1
- $X \subseteq \mathbb{A}$  containing actions that are not urgent in  $Q$

EX:  $a.P \xrightarrow{X}_r \underline{a}.P$  for each  $X \subseteq \mathbb{A}$ , but  $\underline{a}.P \xrightarrow{X}_r \underline{a}.P$  iff  $a \notin X$

We are interested only in steps of the form  $Q \xrightarrow{\mathbb{A}}_r Q'$  (this is a full time step, written  $Q \xrightarrow{1} Q'$ )

- conditional steps can take part in a full time step only in a suitable environment

$\underline{a}.P \xrightarrow{\mathbb{A} \setminus \{a}}_r \underline{a}.P$  can take part in  $\underline{a}.P \parallel_{\{a\}} a.nil \xrightarrow{\mathbb{A}} \underline{a}.P \parallel_{\{a\}} \underline{a}.nil$

# The Functional Behaviour of PAFAS<sub>5</sub>-terms

## Notation:

Initial processes	ranged over by $P, P', \dots$
General processes	ranged over by $Q, Q', \dots, Q_1, \dots$
$\underline{A}_\tau \cup \underline{A}_\tau$	ranged over by $\mu, \mu', \dots, \mu_1, \dots$

$$\text{ACT}_1 \frac{}{\alpha.P \xrightarrow{\alpha} P} \qquad \text{ACT}_2 \frac{}{\underline{\alpha}.P \xrightarrow{\alpha} P} \quad \text{ext. for urgent prefixes}$$

$$\text{READ}_1 \frac{\{\alpha, \underline{\alpha}\} \cap \{\mu_1, \dots, \mu_n\} \neq \emptyset}{\{\mu_1, \dots, \mu_n\} \triangleright Q \xrightarrow{\alpha} \{\mu_1, \dots, \mu_n\} \triangleright Q}$$

as before, but now the read-set  $\{\mu_1, \dots, \mu_n\}$  can also contain urgent actions

$$\text{READ}_2 \frac{Q \xrightarrow{\alpha} Q'}{\{\mu_1, \dots, \mu_n\} \triangleright Q \xrightarrow{\alpha} Q'}$$

$$\text{SYNCH} \frac{\alpha \in A, Q_1 \xrightarrow{\alpha} Q'_1, Q_2 \xrightarrow{\alpha} Q'_2}{Q_1 \parallel_A Q_2 \xrightarrow{\alpha} \text{clean}(Q'_1 \parallel_A Q'_2)}$$

$$\text{PAR} \frac{\alpha \notin A, Q_1 \xrightarrow{\alpha} Q'_1}{Q_1 \parallel_A Q_2 \xrightarrow{\alpha} \text{clean}(Q'_1 \parallel_A Q_2)} + \text{symm}$$

# The Refusal Behaviour of PAFAS<sub>s</sub>-terms

$$\text{NIL}_t \frac{}{\text{nil} \xrightarrow{X}_r \text{nil}}$$

$$\text{ACT}_{t1} \frac{}{\alpha.P \xrightarrow{X}_r \underline{\alpha}.P}$$

$$\text{ACT}_{t2} \frac{\alpha \notin X \cup \{\tau\}}{\underline{\alpha}.P \xrightarrow{X}_r \underline{\alpha}.P} \quad \text{if } \alpha = \tau, \text{ a time step is not possible}$$

$$\text{SUM}_t \frac{Q_i \xrightarrow{X}_r Q'_i \text{ for } i = 1, 2}{Q_1 + Q_2 \xrightarrow{X}_r Q'_1 + Q'_2}$$

$$\text{READ}_t \frac{U(\{\mu_1, \dots, \mu_n\}) \cap (X \cup \{\tau\}) = \emptyset, Q \xrightarrow{X}_r Q'}{\{\mu_1, \dots, \mu_n\} \triangleright Q \xrightarrow{X}_r \{\mu_1, \dots, \mu_n\} \triangleright Q'}$$

$U(\{\mu_1, \dots, \mu_n\})$  denotes the set of urgent actions in  $\{\mu_1, \dots, \mu_n\}$

$$\text{PAR}_t \frac{Q_i \xrightarrow{X_i}_r Q'_i \text{ for } i = 1, 2, X \subseteq (A \cap (X_1 \cup X_2)) \cup ((X_1 \cap X_2) \setminus A)}{Q_1 \parallel_A Q_2 \xrightarrow{X}_r \text{clean}(Q'_1 \parallel_A Q'_2)}$$

# The function $\text{clean}(\_)$

- According to a standard operational semantics

$$(\underline{a} + \underline{b}.a) \parallel_{\{a\}} \underline{a} \xrightarrow{b} Q = a \parallel_{\{a\}} \underline{a}$$

- after the execution of a  $b$ -action, the urgency in the right-hand side becomes **inactive**
- We use the function  $\text{clean}$  to remove such urgencies. EX:

$$\text{clean}(a \parallel_{\{a\}} \underline{a}) = a \parallel_{\{a\}} a$$

$$\text{clean}(b \parallel_{\{a\}} \underline{a}) = b \parallel_{\{a\}} a$$

$$\text{clean}(\underline{a} \parallel_{\{a\}} \underline{a}) = \underline{a} \parallel_{\{a\}} \underline{a}$$

- Notice that

$$(c + b.a) \parallel_{\{a\}} a \xrightarrow{1} \text{clean}((\underline{c} + \underline{b}.a) \parallel_{\{a\}} \underline{a}) = (\underline{c} + \underline{b}.a) \parallel_{\{a\}} \underline{a}$$