Performance evaluation in a Process Algebra with read-actions

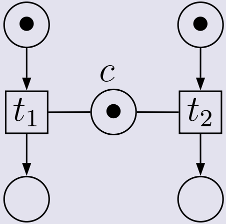
Speaker: Maria Rita Di Berardini[1]

[1]Dip. di Matematica e Informatica, Università di Camerino

Paco Meeting, Lucca 25-26 giugno 2009

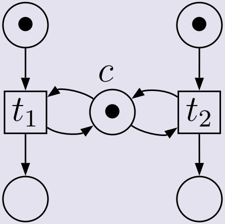
Petri Nets with Read Arcs

- In Petri nets **non-blocking readings** are modelled with a special kind of arcs called **read arcs** (but also condition or test arcs)
- These arcs represent **positive context** conditions: elements needed for an event to occur, but not effected by it



Petri Nets with Read Arcs

- In Petri nets with **ordinary arcs** such a reading must be rendered as a consume/produce loop that causes a loss in concurrency



Read arcs:

1. Allow a faithful representation of systems where the notion of "reading without consuming" is commonly used (databases, concurrent constraint programming, etc.)
2. Allow to specify directly and naturally a level of concurrency greater than in classical nets
3. 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

In this paper

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

Plan

A Basic Process Algebra

The set of processes is generated by

$$P ::= \mathsf{nil} \mid x \mid \alpha.P \mid P + P \mid P\|_A P \mid P[\Phi] \mid \mathsf{rec}\, x.P$$

where $\alpha \in \mathbb{A}_\tau$, $A \subseteq \mathbb{A}$ and Φ is a relabeling function

$$\textsc{Act}\ \frac{}{\alpha.P \xrightarrow{\alpha} P} \qquad \textsc{Sum}\ \frac{P \xrightarrow{\alpha} P'}{P + Q \xrightarrow{\alpha} P'} + \text{sym.}$$

$$\textsc{Synch}\ \frac{\alpha \in A,\, P \xrightarrow{\alpha} P',\, Q \xrightarrow{\alpha} Q'}{P\|_A Q \xrightarrow{\alpha} P'\|_A Q'} \qquad \textsc{Par}\ \frac{\alpha \notin A,\, P \xrightarrow{\alpha} P'}{P\|_A Q \xrightarrow{\alpha} P'\|_A Q} + \text{sym.}$$

other rules are as expected

A Basic Process Algebra

The set of processes is generated by

$$P ::= \mathsf{nil} \mid x \mid \alpha.P \mid \cdots \mid \mathsf{rec}\, x.P \mid \{\alpha_1, \ldots, \alpha_n\} \triangleright P$$

where $\alpha \in \mathbb{A}_\tau$, $A \subseteq \mathbb{A}$, Φ *is a relabeling function, and the read-set* $\{\alpha_1, \ldots, \alpha_n\} \subseteq \mathbb{A}_\tau$.

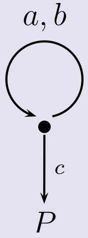
Intuitively

A term like

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

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

$\{a, b\} \triangleright c.P$



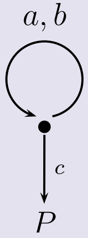
Intuitively

A term like

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

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

$\{a, b\} \triangleright c.P$



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

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

Basic Assumption

- Actions have an **upper time bound** - either 0 or 1 - as a maximal delay. We distinguish between:
  - **patient actions** (time bound 1) $\alpha, \beta, \cdots$
  - **urgent actions** (time bound 0) $\underline{\alpha}, \underline{\beta}, \cdots$

Basic Assumption

- Actions have an **upper time bound** - either 0 or 1 - as a maximal delay. We distinguish between:
  - **patient actions** (time bound 1) $\alpha, \beta, \cdots$
  - **urgent actions** (time bound 0) $\underline{\alpha}, \underline{\beta}, \cdots$
- Patient actions can be delayed for 1 time unit and becomes urgent. Urgent actions (not waiting for a synchronization) have to be performed before time can pass further.

Basic Assumption

- Actions have an **upper time bound** - either 0 or 1 - as a maximal delay. We distinguish between:
  - **patient actions** (time bound 1) $\alpha, \beta, \cdots$
  - **urgent actions** (time bound 0) $\underline{\alpha}, \underline{\beta}, \cdots$
- Patient actions can be delayed for 1 time unit and becomes urgent. Urgent actions (not waiting for a synchronization) have to be performed before time can pass further.

$$
\begin{tikzcd}
a.P \arrow[r, "1"] \arrow[dr, bend right, "a"'] & \underline{a}.P \stackrel{1}{\not\rightarrow} \arrow[d, "a"] \\
& P
\end{tikzcd}
$$

Basic Assumption

- Actions have an **upper time bound** - either 0 or 1 - as a maximal delay. We distinguish between:
  - **patient actions** (time bound 1) $\alpha, \beta, \cdots$
  - **urgent actions** (time bound 0) $\underline{\alpha}, \underline{\beta}, \cdots$
- Patient actions can be delayed for 1 time unit and becomes urgent. Urgent actions (not waiting for a synchronization) have to be performed before time can pass further.

$$
\begin{tikzcd}
a.P \arrow[r, "1"] \arrow[dr, bend right, "a"'] & \underline{a}.P \arrow[d, "a"] \not\xrightarrow{1} \\
& P
\end{tikzcd}
$$

$$
\begin{tikzcd}
\underline{a}.P \parallel_a a.\mathsf{nil} \arrow[r, "1"] \arrow[dr, bend right, "a"'] & \underline{a}.P \parallel_a \underline{a}.\mathsf{nil} \arrow[d, "a"] \not\xrightarrow{1} \\
& P \parallel_a \mathsf{nil}
\end{tikzcd}
$$

Transitional Semantics of PAFAS$_s$

1. Functional Behaviour

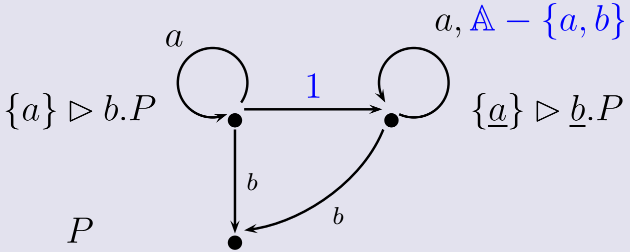
   $Q \xrightarrow{\alpha} Q'$ Q evolves into Q' by performing the action α

2. Refusal Behaviour

   $Q \xrightarrow{X}_r Q'$ It is a conditional time step of duration 1. X is a set of actions that are not urgent and can be refused by Q. These steps can take part in a 'real' time step only in a suitable environment.

   $Q \xrightarrow{1} Q'$ Whenever $Q \xrightarrow{X}_r Q'$ and $X = \mathbb{A}$

An example



Read-actions and Timed behaviour

Example

Consider a variable $V = r.V + w.V$

$$P = (r.o.\mathrm{nil} \parallel_{\{o\}} r.o.\mathrm{nil}) \parallel_{\{r,w\}} V$$

$$\begin{array}{lll} P & \xrightarrow{1} & (\underline{r}.o.\mathrm{nil} \parallel_{\{o\}} \underline{r}.o.\mathrm{nil}) \parallel_{\{r,w\}} (\underline{r}.V + \underline{w}.V) \\ & \xrightarrow{r} & (o.\mathrm{nil} \parallel_{\{o\}} \underline{r}.o.\mathrm{nil}) \parallel_{\{r,w\}} V \\ & \xrightarrow{1} & (\underline{o}.\mathrm{nil} \parallel_{\{o\}} \underline{r}.o.\mathrm{nil}) \parallel_{\{r,w\}} (\underline{r}.V + \underline{w}.V) \\ & \xrightarrow{r} & (\underline{o}.\mathrm{nil} \parallel_{\{o\}} o.\mathrm{nil}) \parallel_{\{r,w\}} V \\ & \xrightarrow{1} & \end{array}$$

Read-actions and Timed behaviour

Example

Consider a variable $V = \{r\} \triangleright w.V$

$$P = (r.o.\mathrm{nil} \parallel_{\{o\}} r.o.\mathrm{nil}) \parallel_{\{r,w\}} V$$

$$\begin{array}{lll}
P & \xrightarrow{1} & (\underline{r}.o.\mathrm{nil} \parallel_{\{o\}} \underline{r}.o.\mathrm{nil}) \parallel_{\{r,w\}} \{\underline{r}\} \triangleright \underline{w}.V \\
 & \xrightarrow{r} & (o.\mathrm{nil} \parallel_{\{o\}} \underline{r}.o.\mathrm{nil}) \parallel_{\{r,w\}} \{\underline{r}\} \triangleright \underline{w}.V \not\xrightarrow{1} \\
 & \xrightarrow{r} & (o.\mathrm{nil} \parallel_{\{o\}} o.\mathrm{nil}) \parallel_{\{r,w\}} \{\underline{r}\} \triangleright \underline{w}.V \\
 & \xrightarrow{1} &
\end{array}$$

Plan

Timing and Fairness

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

An infinite trace $\alpha_0\alpha_1\alpha_2\ldots$ *of* P_0 *is* **fair** *iff there exists a non-Zeno timed execution sequence*

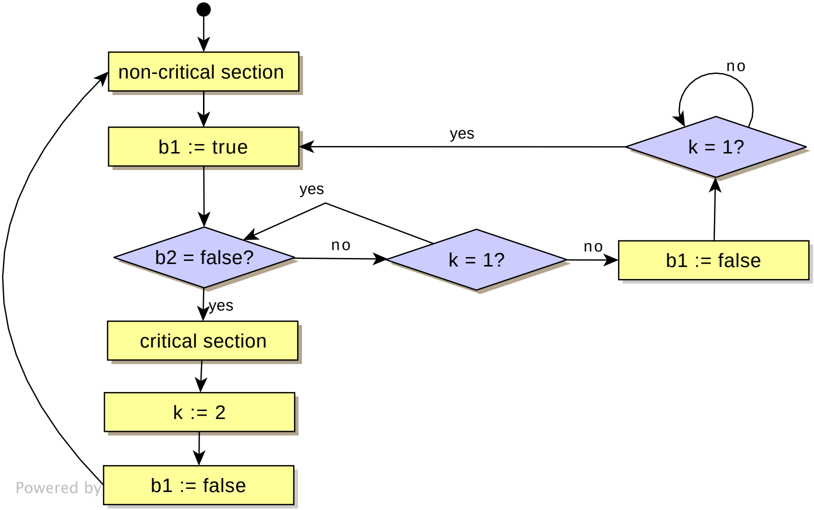
$$P_0 \xrightarrow{1}\xrightarrow{v_0} P_1 \xrightarrow{1}\xrightarrow{v_1} P_2 \cdots P_n \xrightarrow{1}\xrightarrow{v_n} P_{n+1} \cdots$$

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
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).

Plan

Dekker's Algorithm



Powered by

Liveness Property of Dekker's Algorithm

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

D.J. Walker
Automated Analysis of Mutual Exclusion algorithms using CCS
Formal Aspects of Computing **1**, pp. 273-292, 1989

Liveness Property of Dekker's Algorithm

- 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 (may be fairness of actions?)

D.J. Walker
Automated Analysis of Mutual Exclusion algorithms using CCS
Formal Aspects of Computing **1**, pp. 273-292, 1989

Liveness Property of Dekker's Algorithm

- 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 (may be fairness of actions?)
- We have to prevent some kinds of unwanted behaviours as for instance: *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

Variables and non-blocking behaviours

Example

Let :

- $R = r.R$, $W = w.W$,
- $V = r.V + w.V$ and
- $P = (R \parallel_\emptyset W) \parallel_{\{r,w\}} V$

According to our timed operational semantics

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

A run consisting of infinitely many r's is fair

Liveness: a first (negative) result

Definition (Dekker's program variables)

$$\mathrm{B}_1(\mathit{false}) = b_1 \mathit{rf}.\mathrm{B}_1(\mathit{false}) + (b_1 \mathit{wf}.\mathrm{B}_1(\mathit{false}) + b_1 \mathit{wt}.\mathrm{B}_1(\mathit{true}))$$

$$\mathrm{B}_1(\mathit{true}) = b_1 \mathit{rt}.\mathrm{B}_1(\mathit{true}) + (b_1 \mathit{wf}.\mathrm{B}_1(\mathit{false}) + b_1 \mathit{wt}.\mathrm{B}_1(\mathit{true}))$$

$$\ldots\ \ldots$$

Dekker 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

Variables and non-blocking behaviours

Example

Let :

- $R = r.R$, $W = w.W$,
- $V = \{r\} \triangleright w.V$ and
- $P = (R \parallel_{\emptyset} W) \parallel_{\{r,w\}} V$

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

Variables and non-blocking behaviours

Example

Let :

- $R = r.R$, $W = w.W$,
- $V = \{r\} \triangleright w.V$ and
- $P = (R \parallel_{\emptyset} W) \parallel_{\{r,w\}} V$

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

$$
\begin{array}{lll}
P & \xrightarrow{1} & (\underline{r}.R \parallel_{\emptyset} \underline{w}.W) \parallel_{\{r,w\}} \{\underline{r}\} \triangleright \underline{w}.V \\
 & \xrightarrow{r} & (R \parallel_{\emptyset} \underline{w}.W) \parallel_{\{r,w\}} \{\underline{r}\} \triangleright \underline{w}.V \not\xrightarrow{1} \\
 & & \ldots \\
 & \xrightarrow{r} & (R \parallel_{\emptyset} \underline{w}.W) \parallel_{\{r,w\}} \{\underline{r}\} \triangleright \underline{w}.V \\
 & \xrightarrow{w} & P
\end{array}
$$

Variables and non-blocking behaviours

Example

Let :

- $R = r.R$, $W = w.W$,
- $V = \{r\} \triangleright w.V$ and
- $P = (R \parallel_\emptyset W) \parallel_{\{r,w\}} V$

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

$$
\begin{array}{lll}
P & \xrightarrow{1} & (\underline{r}.R \parallel_\emptyset \underline{w}.W) \parallel_{\{r,w\}} \{\underline{r}\} \triangleright \underline{w}.V \\
 & \xrightarrow{r} & (R \parallel_\emptyset \underline{w}.W) \parallel_{\{r,w\}} \{\underline{r}\} \triangleright \underline{w}.V \not\xrightarrow{1} \\
 & & \ldots \\
 & \xrightarrow{r} & (R \parallel_\emptyset \underline{w}.W) \parallel_{\{r,w\}} \{\underline{r}\} \triangleright \underline{w}.V \\
 & \xrightarrow{w} & P
\end{array}
$$

But, writings still block readings

Variables and non-blocking behaviours

Example

Let :

- $R = r.R$, $W = w.W$,
- $V = \{r\} \triangleright w.V$ and
- $P = (R \parallel_\emptyset W) \parallel_{\{r,w\}} V$

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

Variables and non-blocking behaviours

Example

Let :

- $R = r.R,\ W = w.W,$
- $V = \{r\} \triangleright w.V$ and
- $P = (R \parallel_\emptyset W) \parallel_{\{r,w\}} V$

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

A variable without blocking behaviours: $V = \{r, w\} \triangleright \text{nil}$

Some results about liveness of Dekker's

Definition (Dekker's program variables)

$$B_1(\mathit{false}) = \{b_1\mathit{rf}, b_1\mathit{wf}\} \triangleright b_1\mathit{wt}.B_1(\mathit{true})$$

$$B_1(\mathit{true}) = \{b_1\mathit{rt}, b_1\mathit{wt}\} \triangleright b_1\mathit{wf}.B_1(\mathit{false})$$

... ...

- 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, Ex: The two-phase locking protocol

Some results about liveness of Dekker's

Theorem

Dekker is live.

but

Some results about liveness of Dekker's

Theorem

Dekker is live.

but

Theorem

Dekker_ℓ is not live.

where

Some results about liveness of Dekker's

Theorem

Dekker is live.

but

Theorem

$Dekker_\ell$ is not live.

where

Definition ($Dekker_\ell$ program variables)

$$\mathrm{B}_1(\mathit{false}) = \{b_1\mathit{rf}\} \rhd (b_1\mathit{wf}.\mathrm{B}_1(\mathit{false}) + b_1\mathit{wt}.\mathrm{B}_1(\mathit{true}))$$

$$\mathrm{B}_1(\mathit{true}) = \{b_1\mathit{rt}\} \rhd (b_1\mathit{wf}.\mathrm{B}_1(\mathit{false}) + b_1\mathit{wt}.\mathrm{B}_1(\mathit{true}))$$

... ...

Theorem

Dekker is live.

Proof.

Dekker is live iff Dekker_{io} does not have catastrophic cycles.
The absence of catastrophic cycles has been automatically proved with FASE

□

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

A Performance Function

Definition (the performance function)

The performance function $p(Q, O)$ gives the worst-case time the testable process Q needs to satisfy test O:

$$p(Q, O) = \sup\{\zeta(v) \mid v \in \mathrm{DL}(Q \parallel O) \wedge \omega \notin v\}$$

A Performance Function

Definition (the performance function)

The performance function $p(Q, O)$ gives the worst-case time the testable process Q needs to satisfy test O:

$$p(Q, O) = \sup\{\zeta(v) \mid v \in \mathrm{DL}(Q \parallel O) \wedge \omega \notin v\}$$

Definition (a class of users behaviour)

$$U_n = \begin{cases} \underline{in}.\underline{out}.\underline{\omega} & \text{if } n = 1 \\ U_{n-1} \parallel_\omega \underline{in}.\underline{out}.\underline{\omega} & \text{if } n > 1 \end{cases}$$

A Performance Function

Definition (the performance function)

The performance function $p(Q,O)$ gives the worst-case time the testable process Q needs to satisfy test O:

$$p(Q,O) = \sup\{\zeta(v) \mid v \in \mathrm{DL}(Q \parallel O) \wedge \omega \notin v\}$$

Definition (a class of users behaviour)

$$U_n = \begin{cases} \underline{in}.\underline{out}.\underline{\omega} & \text{if } n = 1 \\ U_{n-1} \parallel_{\omega} \underline{in}.\underline{out}.\underline{\omega} & \text{if } n > 1 \end{cases}$$

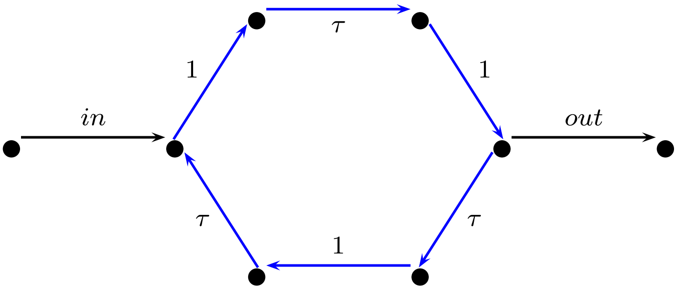
Definition (the response performance function)

$$rp_Q : \mathbf{N} \longrightarrow \mathbf{N}_0 \cup \{\infty\} \text{ such that } rp_Q(n) = p(Q, U_n)$$

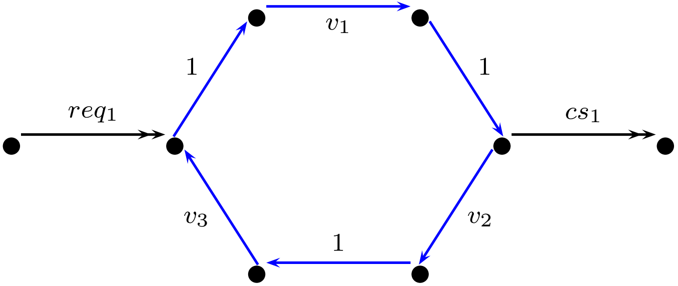
Catastrophic Cycles

Definition (catastrophic 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.



Catastrophic Cycles and Liveness



Conclusions

- We have introduced a 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 the attention