PAFAS at work: Comparing the Worst-Case Efficiency of Three Buffer Implementations *

F. Corradini, M.R. Di Berardini
Dipartimento di Informatica
Università di L’Aquila
{flavio,diberard}@univaq.it

W. Vogler
Institut für Informatik
Universität Augsburg
vogler@informatik.uni-augsburg.de

Abstract

In this paper, we use PAFAS (Process Algebra for Faster Asynchronous Systems) to compare the worst-case efficiency of three bounded-buffer implementations: Fifo, Pipe and Buff. Fifo implements the buffer as a first-in-first-out queue, Pipe implements the buffer as a pipeline queue and Buff implements the buffer as circular queue in an array. We contrast our results with those in [2] and [10] which also aim at comparing the three implementations of the bounded buffer according to some efficiency measure.

1. Introduction

Recently, PAFAS has been proposed as a useful tool for comparing the worst-case efficiency of asynchronous systems [11, 6]. PAFAS is a CCS-like process description language [15] where basic actions are atomic and instantaneous but have associated a time bound interpreted as a maximal time delay for their execution. As discussed in [11, 6], due to these upper time bounds time can be used to evaluate efficiency, but it does not influence functionality (which actions are performed); so compared to CCS, also PAFAS treats the full functionality of asynchronous systems. Processes are compared via a variant of the testing approach developed in [8]. Unlike [8], our tests are not simply test environments but test environments together with a time bound. A process is embedded into the environment (via parallel composition) and satisfies a (timed) test, if success is reached before the time bound in every run of the composed system, i.e. even in the worst case. This gives rise to a preorder relation over processes which is naturally an efficiency preorder. This efficiency preorder can be characterized as inclusion of some kind of refusal traces; this also provides a decidability result for the preorder for finite-state processes. Furthermore, the preorder is independent of the choice to let time progress in a continuous or discrete way; therefore, we only consider discrete time in this paper. These ideas and results were originally successfully studied within the Petri net formalism [16, 10]. We refer the reader to [6] for more details and results on PAFAS.

This paper shows the applicability of PAFAS to concrete meaningful examples. We consider three different implementations of a bounded buffer and relate them according to the above mentioned efficiency preorder. The three implementations are called Fifo, Pipe and Buff. Fifo is a bounded-length first-in-first-out queue, which one could also consider as the specification of a bounded buffer; Pipe is a sequence of one place buffers connected end to end and Buff is an array used in a circular fashion.

We prove that Fifo and Pipe are unrelated according to our (worst-case) efficiency preorder (unrelated means that the former process is not more efficient than the latter one and vice versa); this is presumably in contrast to expectation, since only in Pipe items have to be transported in several steps from one end to the other. Similarly, one would expect Buff to be faster than Pipe, since the latter needs more such steps, but they also turn out to be unrelated. We give good reasons for these results and also prove that Fifo is more efficient than Buff, but not vice versa.

For Buff and Pipe, the same results were obtained in [16], where more or less the same efficiency preorder was defined and studied for Petri nets as specification model instead of a process description language such as PAFAS. This shows that the ideas behind our efficiency preorder are not model-dependent – though, of course, the different models impose a different development.

The same buffers we consider were also contrasted in [2]. Their approach is based on a bisimulation-based preorder; visible actions are regarded as instantaneous and the costs are measured as the number of internal actions. Hence [2] presents an interleaving approach, which disregards the parallel execution of actions. According to this efficiency

* This work was supported by Progetto MURST ‘Saladin: Software Architectures and Languages to Coordinate Distributed Mobile Components’.
measure, it has been proven that \( \text{Fifo} \) is more efficient than \( \text{Buff} \) and \( \text{Buff} \) is more efficient than \( \text{Pipe} \). Since parallel execution of actions is taken into account in the present paper, \( \text{Pipe} \) is incomparable to the others in our approach.

The rest of the paper is organized as follows. The next section briefly recalls PAFAS and the characterization of the testing scenario in terms of refusal traces. Section 3 provides a description of the three buffer implementations and their operational behaviour, while Section 4 studies their relationships according to the efficiency preorder.

### 2. PAFAS and Its Alternative Characterization

This section briefly presents our process description language PAFAS, its operational semantics and the preorder relating processes according to their worst-case efficiency. For a more detailed treatment of the general framework we refer the reader to our original paper [6]. A full version of this paper is [7].

#### 2.1 Processes and Refusal Traces

In [6] a CCS-like process algebra, called PAFAS, is introduced to consider the temporal efficiency of asynchronous systems. To this purpose it is assumed that actions happen within time 1. (See [14] for a similar approach to the temporal behaviour of asynchronous systems.) Since there is no positive lower time bound, components work with intermediate speed even under this assumption - as one expects from asynchronous systems.

PAFAS is a slight extension of the architectural description language described in [12, 13]. It has a CCS [15] sequential composition (called action prefix) and a TCSP [9] parallel composition. In this paper we only report the PAFAS operators strictly needed for our comparison (see [6] for a full account of the language).

We assume that time elapses in a discrete way (in [6] we have shown that our theory is not time domain dependent, meaning that discrete or continuous time give the same semantics of asynchronous systems). Thus, an action prefixed process \( a.P \) can either do action \( a \) and become process \( P \) - as usual in CCS - or can let time 1 pass and become \( a^1.P \); \( a \) is called urgent \( a \), and \( a^1.P \) - as a stand-alone process - cannot let time pass, but can only do \( a \) to become \( P \).

A testing scenario a la De Nicola/Hennessey [8] is developed in [6], where a test is only successfully passed if success is surely reached within a given time bound. As a consequence, we have \( P \triangleq Q \) for the resulting testing preorder \( \triangleq \), if \( P \) serves all possible users or for all patterns of usage as efficiently as \( Q \), i.e. if \( P \) is at least as good as \( Q \) w.r.t. functionality and efficiency. The faster-than-relation \( \sqsupset \) is characterized using some sort of refusal traces which provide decidability of the testing preorder for finite state processes (such as those considered in [12]).

Due to space limitations, we omit the presentation of the respective testing theory and pay more attention to the refusal semantics, considered for our comparison.

\( \mathbb{A} \) (ranged over by \( a, b, c, \ldots \)) is an infinite set of basic actions, which can let time 1 pass before their execution. Action \( \tau \) represents internal activity that is unobservable for other components. Let \( \mathbb{A}_r = \mathbb{A} \cup \{ \tau \} \) (ranged over by \( \alpha, \beta, \ldots \)) and \( \mathbb{A}_r = \{ \alpha \mid \alpha \in \mathbb{A}_r \} \cup \{ \tau \} \) (ranged over by \( \alpha, \beta, \ldots \)) is the set of urgent actions.

\( \mathcal{X} \) is the set of process variables, used for recursive definitions. Elements of \( \mathcal{X} \) are denoted by \( x, y, z, \ldots \).

A general relabelling function is a function \( \Phi : \mathbb{A}_r \rightarrow \mathbb{A}_r \) where the set \( \{ \alpha \in \mathbb{A}_r \mid 0 \neq \Phi^{-1}(\alpha) \neq \{ \alpha \} \} \) is finite and \( \Phi(\tau) = \tau \). As shown in [6], general relabelling functions subsume the classically distinguished operations relabelling and hiding: \( P/A \), where the actions in \( A \) are made internal, is the same as \( P[\Phi_A] \), where the relabelling function \( \Phi_A \) is defined by \( \Phi_A(\alpha) = \tau \) if \( \alpha \in A \) and \( \Phi_A(\alpha) = \alpha \) if \( \alpha \notin A \).

**Definition 2.1 (Timed Processes)** The set \( \mathcal{P} \) of (timed) processes is the set of closed (i.e., without free variables) and guarded (i.e., variable \( x \) in a \( \mu x.P \) only appears within the scope of a prefix \( a.(x) \)), where \( \alpha \in \mathbb{A}_r \) terms generated by the following grammar:

\[
\begin{align*}
P & ::= 0 \mid \gamma.P \mid P + P \mid P[A]P \mid P[\Phi] \mid x \mid \mu x.P
\end{align*}
\]

where \( \gamma \) is \( \alpha \) or \( 0 \) for some \( \alpha \in \mathbb{A}_r \). \( \Phi \) a general relabelling function, \( x \in \mathcal{X} \) and \( A \subseteq \mathbb{A} \) possibly infinite.

0 is the Nil-process, which cannot perform any action, but may let time pass without limit; a trailing 0 will often be omitted, so e.g. \( a.b + c \) abbreviates \( a.b.0 + c.0 \). \( \alpha.P \) and \( \alpha.P \) is (action-) prefixing, known from CCS. In particular, process \( \alpha.P \) performs \( \alpha \) with a maximal delay of 1; hence, it can either perform \( \alpha \) immediately, or can idle for time 1 and become \( \alpha.P \). In the latter case, the idle-time has elapsed; hence \( \alpha \) must either occur or be deactivated (in a choice-context) before time may pass further – unless it has to wait for synchronization with another component (in case \( \alpha \neq \tau \)). This means that our processes are patient: As a stand-alone process, \( \alpha.P \) has no reason to wait; but as a component in \( \alpha.P[A].Q \), it has to wait for synchronization on \( \alpha \) and this can take up to time 1, since component \( a.Q \) may idle this long. \( P_1 + P_2 \) models the choice between two conflicting processes \( P_1 \) and \( P_2 \). \( P_1[A]P_2 \) is the parallel composition of two processes \( P_1 \) and \( P_2 \) that run in parallel and have to synchronize on all actions from \( A \); this synchronization discipline is inspired from TCSP. \( P[\Phi] \) behaves as
$P$ but with the actions changed according to $\Phi$. $\mu e. P$ models a recursive definition; in the examples, we will define processes by recursive equations instead.

We are now ready to define the refusal traces of a process $P$. Intuitively, a refusal trace records, along a computation, which actions process $P$ can perform ($P \xrightarrow{\alpha} P', \alpha \in A_r$) and which actions $P$ can refuse to perform ($P \xrightarrow{\xi} P', X \subseteq A_r$). A transition like $P \xrightarrow{\alpha} P'$ is called a time step.

The actions listed in the set $X$ are not urgent; hence, $P$ is justified in not performing them, but performing a time step instead. Other actions might be urgent, so as a stand-alone-process $P$ might actually be unable to make a time step; but as a component of a larger system, it might take part in a time step if it has to synchronize on those other actions with the environment and the latter can refuse them (see rule $\text{Pref}_{\text{i}2}$ in Fig. 1). $P$ can make a time step in any context, if $X = A$.

**Definition 2.2 (Refusal operational semantics)** The inference rules in Fig. 1 (together with symmetric rules for $\text{Par}_{a1}$ and $\text{Sum}_{a2}$ for actions of $P_2$) define $\Delta_{\text{ref}} \subseteq (P \times P)$, where $X \subseteq A_r$ and $\alpha \subseteq A_r$.

The rules in Fig. 1 explain the operational semantics of PAFAS processes. A process like $\alpha. P$ can either perform action $\alpha$ and then become $P$ (rule $\text{Pref}_{\text{i}3}$), or can let time 1 pass and refuse to perform $\alpha$ (after which action $\alpha$ becomes urgent, rule $\text{Pref}_{\text{i}4}$). A process $P$ prefixed by an urgent action $\alpha. P$, can perform an action $\alpha$ (rule $\text{Pref}_{\text{i}5}$) and on its own cannot delay such an execution (rule $\text{Pref}_{\text{i}6}$). Since internal action $\tau$ never has to be synchronized, a process prefixed by an urgent $\tau$ cannot make a time step. Another rule worth noting is $\text{Par}_{\text{r}}$, which defines which actions a parallel composition can refuse during a time step. The intuition is that $P_1 \parallel A P_2$ can refuse an action $a$ if either $a \not\in A$ ($P_1, P_2$ are not forced to synchronize on $a$) and both $P_1, P_2$ can refuse $a$, or $a \in A$ ($P_1, P_2$ are forced to synchronize on $a$) and either $P_1$ or $P_2$ can refuse $a$. The other rules are as expected.\(^2\)

For sequences $w \in (A_r \cup 2A_r)^*$, we define $P \xrightarrow{w} P'$ as expected: $P \xrightarrow{w} P$ if $w = \varepsilon$ (the empty sequence) or there exist $Q \in \mathcal{P}$ and $\mu \in (A_r \cup 2A_r)$ such that $P \xrightarrow{\mu} Q \xrightarrow{w} P'$ and $w = \mu w'$. Define $P \xrightarrow{v} P'$ if $P \xrightarrow{w} P'$ and $v$ is the sequence $w$ with all $\tau$’s removed. Finally, $\text{RT}(P) = \{w \mid P \xrightarrow{w} P\}$ is the set of refusal traces of $P$.

The efficiency preorder $\sqsubseteq$ is characterized by refusal-trace-inclusion (see [6] for the proof).

**Theorem 2.3 (Characterization of the efficiency preorder)** Process $P$ is faster than $Q$ ($P \sqsubseteq Q$) if and only if $\text{RT}(P) \subseteq \text{RT}(Q)$.

\(^2\)[6] uses a different rule $\text{Rec}_{\text{r}}$ which is equivalent due to guardedness.

---

### 3 Description of the Three Buffers

We now describe three implementations of a buffer of capacity $N + 2$, where $N$ is a fixed positive natural number. The buffer receives and stores values from $V = \{0, 1\}$. We use the following notation: Strings are denoted $s, t, \ldots$; and $|s|$ denotes the length of $s$; thus, $|s| = 0$ means $s = \varepsilon$. $V^k$ denotes the set of strings in $V^*$ of length $k$ while $V_k = \bigcup_{i=0}^k V^i$ denotes the set of strings of length at most $k$.

In the following, we use the notation $\sum_{e \in V} P(e)$, where the “value-variable” $e$ appears in $P(e)$; the notation stands for $P(0) + P(1)$, where $P(0)$ is obtained by substituting $0$ for $e$ in a way that will be obvious, and similarly for $P(1)$.

We also extend $V$ to include values $0 \frac{1}{2}$ and put $D = \{0, 1, \frac{1}{2}\}$ and $D_1 = D \cup \{\frac{1}{2}\}$. $\frac{1}{2}$ is a special value which will denote the absence of values. We assume the properties $0\frac{1}{2} = 0$ and $\frac{1}{2} = \frac{1}{2}$. The extension of $\frac{1}{2}$ to strings is as expected.

#### 3.1 Buffer “Fifo”

Buffer $\text{Fifo}$ directly implements a first-in-first-out queue of capacity $N + 2$; it has no overhead in the form of internal actions, and it is purely sequential. The state is denoted by the string contained in $\text{Fifo}$; thus, the state space is given by $V_{N+2}$. The software architecture of $\text{Fifo}$ is described in Fig. 2. The following formal definition of $\text{Fifo}(s)$ corresponds to the one in [2], while the definition of $\text{Fifo}(s)$ is indigenous to our approach since it describes the process that $\text{Fifo}(s)$ evolves to after a time-step transition. (This applies analogously for the other two implementations below.)

**Definition 3.1 (Buffer $\text{Fifo}$)** Let $s \in V_{N+2}$ and $d \in V$. We define $\text{Fifo} \equiv \text{Fifo}(\varepsilon)$ where $\text{Fifo}(s)$ is defined by the following recursive equations:

1. if $|s| = 0$ (and $s = \varepsilon$), then $\text{Fifo}(s) \equiv \sum_{e \in V} \text{in}(e). \text{Fifo}(e)$
2. if $0 < |s| < N + 2$ and $s = ds'$, then $\text{Fifo}(s) \equiv \text{out}(d). \text{Fifo}(s') + \sum_{e \in V} \text{in}(e). \text{Fifo}(se)$
3. if $|s| = N + 2$ and $s = ds'$, then $\text{Fifo}(s) \equiv \text{out}(d). \text{Fifo}(s')$

The target state for a time step of $\text{Fifo}(s)$ is denoted by $\text{Fifo}(s')$ and is defined by the following recursive definitions:

1. if $|s| = 0$ (and $s = \varepsilon$), then $\text{Fifo}(s) \equiv \sum_{e \in V} \text{in}(e). \text{Fifo}(e)$
2. if $0 < |s| < N + 2$ and $s = ds'$, then $\text{Fifo}(s) \equiv \text{out}(d). \text{Fifo}(s') + \sum_{e \in V} \text{in}(e). \text{Fifo}(se)$
3. if $|s| = N + 2$ and $s = ds'$, then $\text{Fifo}(s) \equiv \text{out}(d). \text{Fifo}(s')$
For $Fifo$ processes, the transitions of the processes are:

1. $|s| < N + 2$ implies $Fifo(s) \xrightarrow{\text{in}(d)} Fifo(sd)$;
2. $|s| > 0$ and $s = ds'$ implies $Fifo(s) \xrightarrow{\text{out}(d)} Fifo(s')$;
3. $Fifo(s) \xrightarrow{\text{out}} Fifo(s)$;
4. $|s| < N + 2$ implies $Fifo(s) \xrightarrow{\text{in}(d)} Fifo(sd)$;
5. $|s| > 0$ and $s = ds'$ implies $Fifo(s) \xrightarrow{\text{out}(d)} Fifo(s')$;
6. $|s| = 0$ (i.e. $s = e$) implies $Fifo(s) \xrightarrow{\text{out}} Fifo(s)$, for every $Y \subseteq \mathbb{A} \setminus \{\text{out}(d), \text{in}(0), \text{in}(1)\}$;
7. $0 < |s| < N + 2$ and $s = ds'$ implies $Fifo(s) \xrightarrow{\text{out}} Fifo(s)$, for every $Y \subseteq \mathbb{A} \setminus \{\text{out}(d), \text{in}(0), \text{in}(1)\}$;
8. $|s| = N + 2$ and $s = ds'$ implies $Fifo(s) \xrightarrow{\text{out}} Fifo(s)$, for every $Y \subseteq \mathbb{A} \setminus \{\text{out}(d)\}$.

Observe that $Fifo(s)$ for $0 < |s| < N + 2$ has an urgent input and an urgent output to perform; after each it becomes some $Fifo(s')$, which has no urgent action. This reflects that a sequential process may let time 1 pass after each action.

### 3.2 Buffer “Pipe”

A buffer can also be seen as the “concatenation” of cells, each of them containing at most one value. A cell is an input-output device. It is defined as follows:

#### Definition 3.3 (Cells)

1. $\mathcal{C}(\bot) \equiv \sum_{e \in \mathbb{V}} in(e).\mathcal{C}(e)$ denotes the empty cell. It can only input either value 0 or 1;
2. $\mathcal{C}(d) \equiv out(d).\mathcal{C}(\bot)$ denotes a cell containing the value $d \in \mathbb{V}$. It can only output value $d$;
3. $\mathcal{C}(\bot) \equiv \sum_{e \in \mathbb{V}} in(e).\mathcal{C}(e)$ denotes the empty cell after a time-step;

4. \( C(d) \equiv \text{out}(d), C(\bot) \) denotes the cell containing a value \( d \) after a time-step.

The special value \( \bot \), contained in the empty cell, denotes the absence of values in the cell. A buffer of capacity \( N+2 \) can be obtained by connecting \( N+2 \) cells end to end and relabelling communication channels appropriately. Every value pushed into the buffer crosses every cell before reaching the last one from where the value can be popped. The actions \( \delta_j(d), d \in V \), denote the sending of value \( d \) from the \( j+1 \)th to the \( j \)th cell. The software architecture of Pipe is described in Fig. 3.

**Definition 3.4 (Buffer Pipe)** Let \( x \in D_\bot \) and \( j = 0, \ldots, N+1 \). The \( j \)-th cell of Pipe is defined by \( C_j(x) \equiv C(x)[\Phi_j] \), where the relabelling \( \Phi_j \) is defined as:

\[
\Phi_0(\alpha) = \begin{cases} 
\delta_0(0) & \text{if } \alpha = \text{in}(0) \\
\delta_0(1) & \text{if } \alpha = \text{in}(1) \\
\alpha & \text{otherwise}
\end{cases}
\]

\[
\Phi_N(\alpha) = \begin{cases} 
\delta_N(0) & \text{if } \alpha = \text{out}(0) \\
\delta_N(1) & \text{if } \alpha = \text{out}(1) \\
\alpha & \text{otherwise}
\end{cases}
\]

and for \( j = 1, \ldots, N, \)

\[
\Phi_j(\alpha) = \begin{cases} 
\delta_j(0) & \text{if } \alpha = \text{in}(0) \\
\delta_j(1) & \text{if } \alpha = \text{in}(1) \\
\delta_j(0) & \text{if } \alpha = \text{out}(0) \\
\delta_j(1) & \text{if } \alpha = \text{out}(1) \\
\alpha & \text{otherwise}
\end{cases}
\]

Now, let \( A_j = \{\delta_j(0), \delta_j(1)\} \), for every \( j = 0, \ldots, N \), and \( A = \bigcup_{j=0}^N A_j \). For any string \( s = s_0 \ldots s_{N+1} \in D^{N+2}_\bot \), define

\[
I(s) \equiv C_0(s_0) \parallel A_0 \parallel C_1(s_1) \parallel A_1 \ldots \parallel A_N \parallel C_{N+1}(s_{N+1})
\]

Finally, we define Pipe as Pipe\(_{\bot}^{N+2}\) where Pipe\(_s\) in turn is defined as \( I(s)/A \), i.e. it is obtained by abstracting from internal details. Recall that \( I(s)/A \equiv I(s)[\Phi_A] \), where the relabelling function \( \Phi_A \) is defined as \( \Phi_A(\alpha) = \tau \) if \( \alpha \in A \) and \( \Phi_A(\alpha) = \alpha \) if \( \alpha \notin A \).

The following two propositions describe the behaviour of Pipe.

**Proposition 3.5** Let \( s = s_0 \ldots s_{N+1} \in D^{N+2}_\bot \) and \( d \in V \). Then, all action transitions of the process Pipe\(_s\) are the following:

1. \( s_{N+1} \in \{\bot, \bot\} \) implies Pipe\(_s\) \( \xrightarrow{\text{out}(d)} \) Pipe\(_{s'}\), where \( s' = s_0 \ldots s_{N} \).

2. \( s_0 \in \{d, \bot\} \) implies Pipe\(_s\) \( \xrightarrow{\text{out}(d)} \) Pipe\(_{s'}\), where \( s' = s_1 \ldots s_{N+1} \);

3. if there exists \( j \in \{0, \ldots, N\} \) such that \( s_j \in \{\bot, \bot\} \) and \( s_{j+1} \in \{d, \bot\} \) then Pipe\(_s\) \( \xrightarrow{\delta_j} \) Pipe\(_{s'd} \), where \( s'' = s_0 \ldots s_{j-1} \) (which is \( \varepsilon \) for \( j = 0 \)) and \( s'' = s_{j+2} \ldots s_{N+1} \) (which is \( \varepsilon \) for \( j = N \)).

For the result showing how Pipe\(_s\) evolves by performing time steps, i.e. \( \delta \)-transitions, recall the definition of \( \delta \) for string \( s \) given at the beginning of this section.

**Proposition 3.6** Let \( s = s_0 \ldots s_{N+1} \in D^{N+2}_\bot \). If \( s_0 = \bot \) and \( s_{j+1} = \delta_j \) for some \( j = 0, \ldots, N \) and \( d \in V \), then Pipe\(_s\) \( \xrightarrow{\delta_j} \) Pipe\(_{s'}\), for every \( X \subseteq A \). Otherwise, the time steps of Pipe\(_s\) are the following, where \( a \in V \cup \{\bot\} \):

1. \( s = a s' \bot \) implies Pipe\(_s\) \( X \rightarrow_r \) Pipe\(_{s'}\), for every \( X \subseteq A \setminus \{\text{out}(a), \text{in}(0), \text{in}(1)\} \) (which is \( A \setminus \{\text{in}(0), \text{in}(1)\} \) for \( a = \bot \));

2. \( s = a s' \bot \) implies Pipe\(_s\) \( X \rightarrow_r \) Pipe\(_{s'}\), for every \( X \subseteq A \setminus \{\text{out}(a)\} \) (which is \( A \) for \( a = \bot \));

3. \( s = a s' \bot \) implies Pipe\(_s\) \( X \rightarrow_r \) Pipe\(_{s'}\), for every \( X \subseteq A \setminus \{\text{in}(0), \text{in}(1)\} \);

4. \( s = a s' \bot \) implies Pipe\(_s\) \( X \rightarrow_r \) Pipe\(_{s'}\), for every \( X \subseteq A \).

**3.3 Buffer “Buff”**

Assume that \( N \) cells are not connected end to end (as in Pipe) but are used as a storage. The cells interact with a centralized buffer controller that can store two more values. The software architecture of Buff is described in Fig. 4. First, we describe the functional behaviour of store Mem.

**Definition 3.7 (Mem)** Let \( x \in D_\bot \) and \( j = 0, \ldots, N - 1 \). The \( j \)-th element of Mem is described by process \( B_j(x) \equiv C(x)[\Phi_j'] \), where the various relabelling functions are defined as:

\[
\Phi_j'(\alpha) = \begin{cases} 
\omega_j(0) & \text{if } \alpha = \text{in}(0) \\
\omega_j(1) & \text{if } \alpha = \text{in}(1) \\
\rho_j(0) & \text{if } \alpha = \text{out}(0) \\
\rho_j(1) & \text{if } \alpha = \text{out}(1) \\
\alpha & \text{otherwise}
\end{cases}
\]

Let \( s = s_0 \ldots s_{N-1} \in D^N \). Define

\[
\text{Mem}(s) \equiv B_0(s_0) \parallel \ldots \parallel B_{N-1}(s_{N-1})
\]

and let \( B = \{\omega_j(d), \rho_j(d) \mid d \in V, j = 0, \ldots, N - 1\} \).
Mem acts as a store. It is used by a buffer controller (BC) to store data received from the external environment. BC uses the $N$ cells of Mem as a circular queue (ordered as $0 < 1 < \ldots < N - 1$). More in detail, the buffer controller accepts a value from the external environment and then writes the accepted value into the first available empty cell. It cannot accept any other value until the accepted one is actually stored in one of the $N$ cells. BC also retains the oldest undelivered value and delivers it whenever possible.

The state $BC(x, y, i, m)$ of BC is determined by the four arguments:

1. $x \in V$, the value in input. That is, the value that BC has recently accepted from the environment; if $x = \bot$, then a new value can be accepted.

2. $y \in V$, the value in output. That is, the value read from Mem that can be made available to the external environment; if $y = \bot$, then no value in output is available.

3. $i$, the index of the cell containing the oldest undelivered message.

4. $m$, the number of messages in store Mem.

Since the messages are stored in contiguous cells of the circular queue, the first empty cell is given by $(i + m) \mod N$, i.e. the sum of $i$ and $m$ modulo $N$. Finally, if $n$ denotes the number of messages in the buffer, we always have $m \leq n \leq m + 2$.

BC can accept a new value from the environment only if $x = \bot$; on the contrary, if $x = d \in V$, then BC can only write $d$ into the first available cell of Mem (i.e. perform $\omega(i + m) (d)$). This storing operation is possible if $m < N$.

Similarly, BC can deliver a value only if $y = a \in V$. Again, if $y = \bot$, then BC can only read the oldest undelivered value from Mem (i.e. perform $\rho_i(a)$). This operation is possible, if Mem is not empty, that is $m > 0$.

The buffer controller is then defined as follows.

**Definition 3.8 (Buffer Controller)** Let $d, a \in V$, $0 \leq i \leq N - 1$ and $0 \leq m \leq N$.

1. $BC(\bot, \bot, i, 0) \equiv \sum_{d \in V} in(d).BC(d, \bot, i, 0)$;

2. $m > 0$ implies
   $BC(\bot, \bot, i, m) \equiv (\sum_{d \in V} in(d).BC(d, \bot, i, m)) + (\sum_{a \in V} \rho_i(a).BC(\bot, a, (i + 1) \mod N, m - 1))$;
3. $BC(d, \perp, i, 0) \equiv \omega_i(d).BC(\perp, \perp, i, 1)$;

4. $0 < m < N$ and $j = (i + m) \mod N$ imply
   $BC(d, \perp, i, m) \equiv \omega_j(d).BC(\perp, \perp, i, m + 1) + (\sum_{a \in V} \rho_i(a).BC(d, a, (i + 1) \mod N, m - 1));$

5. $BC(d, \perp, i, N) \equiv \sum_{a \in V} \rho_i(a).BC(d, a, (i + 1) \mod N, N - 1);$  

6. $BC(\perp, a, i, m) \equiv (\sum_{a \in V} \rho_i(a).BC(d, a, i, m)) + out(a).BC(\perp, \perp, i, m);$  

7. $m < N$ and $j = (i + m) \mod N$ imply
   $BC(d, a, i, m) \equiv \omega_j(d).BC(\perp, a, i, m + 1) + out(a).BC(d, \perp, i, m);$  

8. $BC(d, a, i, N) \equiv out(a).BC(d, \perp, i, N).$

Denote by $BC(x, y, i, m)$ the target process of a step-time out of $BC(x, y, i, m)$.

1. $BC(\perp, \perp, i, 0) \equiv \sum_{a \in V} \sum_{d \in \mathbb{D}} . in(d).BC(d, \perp, i, 0).$

2. $0 < m$ implies $BC(\perp, \perp, i, m) \equiv \sum_{a \in V} \sum_{d \in \mathbb{D}} . in(d).BC(d, \perp, i, m) + (\sum_{a \in V} \rho_i(a).BC(\perp, a, (i + 1) \mod N, m - 1));$

3. $BC(d, \perp, i, 0) \equiv \omega_i(d).BC(\perp, \perp, i, 1).$

4. $0 < m < N$ and $j = (i + m) \mod N$ imply
   $BC(d, \perp, i, m) \equiv \omega_j(d).BC(\perp, \perp, i, m + 1) + (\sum_{a \in V} \rho_i(a).BC(d, a, (i + 1) \mod N, m - 1));$

5. $BC(d, \perp, i, N) \equiv \sum_{a \in V} \rho_i(a).BC(d, a, (i + 1) \mod N, N - 1).$

6. $BC(\perp, a, i, m) \equiv (\sum_{a \in V} \sum_{d \in \mathbb{D}} . in(d).BC(d, a, i, m)) + out(a).BC(\perp, \perp, i, m).$

7. $m < N$ and $j = (i + m) \mod N$ imply
   $BC(d, a, i, m) \equiv \omega_j(d).BC(\perp, a, i, m + 1) + out(a).BC(d, \perp, i, m).$

8. $BC(d, a, i, N) \equiv out(a).BC(d, \perp, i, N).$

Observe that process $BC$ works sequentially; in particular, $BC(\perp, a, i, m)$ has urgent in- and output actions, but may let time 1 pass after each of them.

Process $Buff$ is the parallel composition of the memory $Mem$ and the buffer controller $BC$, which synchronize on actions in $B$, i.e. all actions of $Mem$. Thus, $Buff$ works sequentially as $BC$ does.

**Definition 3.9 (Buff)** We define $Buff \equiv Buff(\perp, \perp, \perp, 0, 0)$ based on the following: Let $x, y \in V \cup \{\perp\}, s = s_0 \ldots s_{N-1} \in D_N^x$, $0 \leq i \leq N - 1$ and $0 \leq m \leq N$ satisfy the buff-invariant that $s_j \in \{\perp, \perp\}$ iff $j \not\in \{i, i+1, \ldots, i+m-1\}$ (viewed mod $N$ and empty if $m = 0$). Define

$Buff(s, x, y, i, m) \equiv (Mem(s) || B(BC(x, y, i, m))) / B$

$Buff(s, x, y, i, m) \equiv (Mem(s) || B(BC(x, y, i, m))) / B.$

Observe that $Buff(\ldots)$ is a process where all components – in particular $BC$ – have urgent actions to perform, independently of $s$. In $Buff(\ldots)$, this is not the case for $BC$, but it might be the case for some components of $Mem$; this is indicated by underlined items in $s$. The following proposition states transitional properties of the $Buff$-processes.

**Proposition 3.10** Let $a, d \in V$ and $s, i, m$ as in 3.9. All action transitions of $Buff(s, d, a, i, m)$ are the following, where in particular the buff-invariant is preserved:

1. $Buff(s, \perp, \perp, i, m) \Rightarrow Buff(s, d, \perp, i, m);$

2. $m > 0$ implies $Buff(s, \perp, \perp, i, m) \Rightarrow Buff(s', \perp, a, (i + 1) \mod N, m - 1), if s_i \in \{a, \perp\}$ and $s' = s_0 \ldots \hat{s}_i \ldots s_{N-1};$

3. $m > 0$ implies $Buff(s, d, \perp, i, m) \Rightarrow Buff(s', d, a, (i + 1) \mod N, m - 1), if s_i \in \{a, \perp\}$ and $s' = s_0 \ldots \hat{s}_i \ldots s_{N-1};$

4. $m < N$ implies $Buff(s, d, \perp, i, m) \Rightarrow Buff(s', \perp, \perp, i, m + 1), if j = (i + m) \mod N and s = s_0 \ldots s_{j-1} \hat{s}_j \ldots s_{N-1};$

5. $Buff(s, d, a, i, m) \Rightarrow Buff(s, d, a, i, m);$

6. $Buff(s', d, a, i, m) \Rightarrow Buff(s', d, a, i, m);$

7. $Buff(s, d, a, i, m) \Rightarrow Buff(s, d, a, i, m);$

8. $Buff(s', d, a, i, m) \Rightarrow Buff(s', d, a, i, m);$

The following proposition deals with $Buff$ in place of $Buff$.

**Proposition 3.11** Let $a, d \in V$ and $s, i, m$ as in 3.9. All action transitions of $Buff(s, d, a, i, m)$ are the following, where in particular the buff-invariant is preserved:

1. $Buff(s, \perp, \perp, i, m) \Rightarrow Buff(s', d, \perp, i, m);$

2. $m > 0$ implies $Buff(s, \perp, \perp, i, m) \Rightarrow Buff(s', a, (i + 1) \mod N, m - 1), if s_i \in \{a, \perp\}$ and $s' = s_0 \ldots s_{j-1} \hat{s}_j \ldots s_{N-1};$
3. \( m > 0 \) implies \( \text{Buff}(s, d, \perp, i, m) \xrightarrow{\tau_r} \text{Buff}(s', d, a, (i + 1) \mod N, m - 1) \), if \( s_i \in \{a, \perp\} \) and 
\( s' = s_0 \ldots s_{i-1} \perp s_{i+1} \ldots s_{N-1}; \)

4. \( m < N \) implies \( \text{Buff}(s, d, \perp, i, m) \xrightarrow{\tau_r} \text{Buff}(s', d, a, (i + 1) \mod N, m + 1) \), if \( j = (i + m) \mod N \) and 
\( s' = s_0 \ldots s_{i-1}d s_{i+1} \ldots s_{N-1}; \)

5. \( \text{Buff}(s, \perp, a, i, m) \xrightarrow{\text{in}(d)} \text{Buff}(s, d, a, i, m); \)

6. \( \text{Buff}(s, \perp, a, i, m) \xrightarrow{\text{out}(a)} \text{Buff}(s, \perp, \perp, i, m); \)

7. \( \text{Buff}(s, d, a, i, m) \xrightarrow{\text{out}(a)} \text{Buff}(s, d, \perp, i, m); \)

8. \( m < N \) implies \( \text{Buff}(s, d, a, i, m) \xrightarrow{\tau_r} \text{Buff}(s', d, a, (i + 1) \mod N, m + 1) \), if \( j = (i + m) \mod N \) and 
\( s' = s_0 \ldots s_{i-1}d s_{i+1} \ldots s_{N-1}. \)

Finally, we state how the Buff-processes evolve by performing \( \xrightarrow{\tau_r} \)-transitions.

**Proposition 3.12** Let \( a, d \in V \), \( X \subseteq A \) and \( x, y, s, i, m \) as in 3.9. All the time-steps of the Buff-processes are the following.

1. \( \text{Buff}(s, x, y, i, m) \xrightarrow{X} \text{Buff}(s, x, y, i, m); \)

2. \( m > 0 \) implies \( \text{Buff}(s, \perp, \perp, i, m) \xrightarrow{X} \text{Buff}(s, \perp, \perp, i, m); \)

3. \( \text{Buff}(s, \perp, a, i, 0) \xrightarrow{Y} \text{Buff}(s, \perp, \perp, i, 0), \) for every \( Y \subseteq A \setminus \{\text{in}(0), \text{in}(1)\}. \)

4. \( \text{Buff}(s, \perp, a, i, m) \xrightarrow{Y} \text{Buff}(s, \perp, a, i, m), \) for every \( Y \subseteq A \setminus \{\text{out}(a), \text{in}(0), \text{in}(1)\}. \)

5. \( \text{Buff}(s, d, \perp, i, m) \xrightarrow{X} \text{Buff}(s, d, \perp, i, m); \)

6. \( 0 \leq m < N \) implies \( \text{Buff}(s, d, a, i, m) \xrightarrow{X} \text{Buff}(s, d, a, i, m), \) for every \( Y \subseteq A \setminus \{\text{out}(a)\}. \)

7. \( \text{Buff}(s, d, a, i, N) \xrightarrow{Y} \text{Buff}(s, d, a, i, N), \) for every \( Y \subseteq A \setminus \{\text{in}(0)\}. \)

4 Comparing the three buffers

This section is the core of the paper. We compare the three buffer implementations with respect to the efficiency preorder \( \sqsubseteq \) discussed in Section 2.1. To do this, we will exploit the alternative characterization in terms of refusal traces.

Given two processes \( P \) and \( Q \), to prove that \( P \) is not more efficient than \( Q \), \( P \not\sqsubseteq Q \), we exhibit a refusal trace of \( P \) that \( Q \) cannot perform. This is sufficient since, by Theorem 2.3, \( \text{RT}(P) \not\subseteq \text{RT}(Q) \) implies \( P \not\sqsubseteq Q \). On the other hand, to prove that \( P \sqsubseteq Q \), we show that \( \text{RT}(P) \subseteq \text{RT}(Q) \) again by Theorem 2.3. To prove this turns out to be a little bit more involved. It is well-known, however, that trace inclusion can be shown by exhibiting a suitable simulation relation. In our setting, to prove \( \text{RT}(P) \subseteq \text{RT}(Q) \), we give a simulation relation between states of the refusal transitional semantics of \( P \) and of \( Q \). A simulation relation is defined as follows:

A relation \( R \) is a simulation relation for two processes \( P \) and \( Q \) if \( (P, Q) \in R \) and whenever \( (R, S) \in R \) and \( R \xrightarrow{\mu} R' \), where either \( \mu = \alpha \in \mathbb{A}_r \) or \( \mu = X \subseteq A \), then either \( S \xrightarrow{\alpha} S' \) or \( S \xrightarrow{\tau} S' \) if \( \mu = \tau \) and in either cases \( (R', S') \in R \).

The existence of a simulation relation for two processes \( P \) and \( Q \) ensures \( \text{RT}(P) \subseteq \text{RT}(Q) \) and hence \( P \sqsubseteq Q \).

4.1 Relating Fifo and Pipe

We start with \( \text{Fifo} \equiv \text{Fifo}(\varepsilon) \) and \( \text{Pipe} \equiv \text{Pipe}(\perp^{N+2}) \) (for our fixed positive \( N \)) and prove that they are unrelated.

**Theorem 4.1** 1. \( \text{Pipe} \not\sqsubseteq \text{Fifo} \), and

2. \( \text{Fifo} \not\sqsubseteq \text{Pipe} \).

**Proof:** (Sketch) Item 1. \( \text{Pipe} \) can perform the refusal trace \( v_1 = \{\text{in}(0)\} \) while \( \text{Fifo} \) cannot. Hence, \( \text{RT}(\text{Pipe}(\perp^{N+2})) \not\subseteq \text{RT}(\text{Fifo}(\varepsilon)) \).

Item 2. \( \text{Fifo} \) can perform the refusal trace \( v_2 = \{\text{in}(0)\} \) while \( \text{Pipe} \) cannot. Hence, \( \text{RT}(\text{Fifo}(\varepsilon)) \not\subseteq \text{RT}(\text{Pipe}(\perp^{N+2})) \).

Theorem 4.1 shows that the worst-case behaviours of \( \text{Fifo} \) and \( \text{Pipe} \) are unrelated. The intuition behind these results is:

- Besides input and output of values, process \( \text{Pipe} \) must perform internal activities in order to manage the queue of cells (i.e. to move values from a cell to the next one). If \( \text{Pipe} \) receives a value \( d \), then this value must pass through the \( N + 2 \) cells before being delivered, that is before an \( \text{out}(d) \)-action becomes available, and each move might take time 1. This is also the case, if \( d \) is the only value in \( \text{Pipe} \). In this situation, \( \text{Fifo} \) would be ready to deliver \( d \) right after \( d \) is received and time 1 has passed.

- Conversely, after the execution of an \( \text{in-} \) or \( \text{out-} \) action, \( \text{Fifo} \) reaches a state in which it can let time pass in any context (see above). This is not necessarily true for \( \text{Pipe} \): for example, \( \text{Pipe}(\perp^{N+1}) \) can perform \( \text{out}(0) \) and reach a state \( P' \equiv \text{Pipe}(\perp^{N+1}) \) in which the \( \text{in-} \) actions are urgent. Hence, in state \( P' \), buffer \( \text{Pipe} \) cannot perform an unconditional time step while \( \text{Fifo} \), in the corresponding state \( \text{Fifo}(\varepsilon) \), can.
More intuitively speaking, Pipe is a distributed implementation, where in particular input and output are independent activities; in FIFO, these are sequential, they block each other. This point, and its effect on the efficiency, might be easily overlooked when comparing FIFO and Pipe without a formal treatment.

4.2 Relating FIFO and Buff

Now we relate Buff \( \equiv \text{Buff}(\bot^N, \bot, \bot, 0, 0) \) and FIFO for our fixed positive \( N \).

**Theorem 4.2**

1. Buffer \( \not\equiv \) FIFO

2. FIFO \( \not\equiv \) Buff

**Proof:** (Sketch) Item 1. Buffer can perform the refusal trace \( v = \text{in}(0)\emptyset\{\text{out}(0)\} \) while FIFO cannot.

Item 2. We present a simulation relation relating FIFO(\( \varepsilon \)) and Buff(\( \bot^N, \bot, \bot, 0, 0 \)). This implies RT(FIFO(\( \varepsilon \))) \( \subseteq \) RT(Buff(Buff(\( \bot^N, \bot, \bot, 0, 0 \))).

We define a function \( g : D_1^N \times \{0, \ldots, N - 1\} \times \{0, \ldots, N\} \rightarrow \mathcal{V}_N \) that extracts from the Mem-content \( s \in D_1^N \) (we take to be \( s = s_0 \ldots s_{N-1} \)) \( m \in \{0, \ldots, N\} \) items beginning with the \( t \)th for \( i \in \{0, \ldots, N - 1\} \); this extraction removes underlines, hence:

\[
g(s, i, m) = \varepsilon \text{ if } m = 0, g(s, i, m) = x_1 \text{ if } m = 1 \text{ and } g(s, i, m) = x_i x_{(i+1) \bmod N} \ldots x_{(i+m-1) \bmod N} \text{ otherwise, where } x_j = d \text{ if } s_j \in \{d, \bot\} \text{ for } j = 0, \ldots, N - 1.
\]

The simulation relation consists of the following pairs for \( i \in \{0, \ldots, N\}, t \in \mathcal{V}_N, s \in D_1^N \) and \( g(s, i, |t|) = t \):

a) for \( s \in \{\bot, \bot\}^N \), \( \text{FIFO}(\varepsilon), \text{Buff}(s, \bot, \bot, i, 0) \) and \( \text{Buff}(s, \bot, \bot, i, 0) \)

b) for \( \text{Buff}(s, \bot, \bot, i, 0) \) and \( \text{FIFO}(at), \text{Buff}(s, \bot, a, i, |t|) \text{ and } \text{FIFO}(at), \text{Buff}(s, \bot, a, i, |t|) \)

c) for \( |t| = N \), \( \text{FIFO}(atd), \text{Buff}(s, d, a, i, N) \) and \( \text{FIFO}(atd), \text{Buff}(s, d, a, i, N) \)

We now give the intuition behind these formal results, which are as one would presumably expect.

- Buff, like Pipe, performs internal activities to manage the store. Hence, it cannot be more efficient than FIFO.

- Although Buff is a distributed implementation (as Pipe is), it works sequentially (see above). Therefore, we do not have the effect that prevents FIFO from being more efficient than Pipe, and FIFO is indeed strictly more efficient than Buff.

4.3 Relating Pipe and Buff

Finally, we relate Pipe and Buff. Again, one would expect that Buff is more efficient, because it takes less time to move an item from input to output, but actually these two buffer implementations are unrelated in general – as the following theorem shows.

**Theorem 4.3**

1. Pipe \( \not\equiv \) Buff, provided \( N > 1 \),

2. Pipe \( \not\equiv \) Buff, provided \( N = 1 \),

3. Buff \( \not\equiv \) Pipe in any case.

**Proof:** Item 1. For \( X = \{\text{out}(0)\} \), Pipe can perform the refusal trace \( v = \text{in}(0)\emptyset\{\text{out}(0)\} \) while Buff cannot.

Item 2. Consider the relation composed by the pairs (Pipe(\( s_0 s_1 s_2 \)), Buff(\( s_1, x_2, x_0, 0, m \)) and (Pipe(x_0 x_1 x_2), Buff(\( s_1, x_2, x_0, 0, m \)), where \( s_i \in D_1 \) and \( x_i \in \mathcal{V} \cup \{\bot\} \) are such that \( s_i \in \{x_1, x_2\} \) and either \( s_1 \in \{\bot, \bot\} \) and \( m = 0 \) or \( s_1 \notin \{\bot, \bot\} \) and \( m = 1 \). This simulation contains (Pipe(\( \bot^3 \)), Buff(\( \bot, \bot, \bot, 0, 0 \))).

Item 3. If we had Buff \( \not\equiv \) Pipe then, by Theorem 4.2.2 and transitivity of \( \not\equiv \), we would have FIFO \( \equiv \) Pipe. This would contradict Theorem 4.1.2.

- Buff, similarly to Pipe, performs internal actions to manage the store, namely for each item one for putting the item into the array and one for taking it out. For Pipe, this number is larger in case \( N > 1 \), explaining why Pipe cannot be more efficient than Buff.

- For \( N = 1 \), the array of Buff degenerates to a cell for one item; hence, Pipe and Buff have a very similar architecture. But in Pipe, input and output are independent, while in Buff they block each other, as explained above. Thus, Pipe is faster than Buff in this case.

- This independence of input and output in Pipe always prevents Buff from being more efficient.

5 Concluding Remarks

In this paper, we have shown the applicability of PAFA to concrete meaningful examples by contrasting the worst-case efficiency of three bounded-buffers.

The worst-case efficiency is one of the most prominent efficiency measures in the traditional theory of computation and complexity. This paper shows how such an efficiency measure can be successfully and smoothly applied to compare complex systems.

Several papers in the literature have advocated the importance of making qualitative and quantitative analysis of complex systems early in the software life cycle stages. This explains the recent proliferation of papers aiming at comparing systems according to non qualitative aspects. Most of them concentrate on performance evaluation in the traditional sense allowing for analysis like throughput, response time, component utilization etc. Typically they derive performance models such as queuing networks.
or Markov chains (see [1] and [3] resp., and references therein) from the dynamic behaviour of the system description. Over such performance models, performance analysis as mentioned above can be carried out by exploiting standard techniques.

Our framework allows for the integrated study of both functionality and efficiency of the system dynamic behaviour. Besides comparing different system descriptions according to these two aspects (as we have done in the current paper), the framework is also particularly suitable during the step-wise refinement of abstract specifications. In [6] we consider a system server that manages requests from the external environment. We start with a purely sequential specification and then refine, step-by-step, this very abstract description by adding more and more parallelism to reach a completely parallel description. This latter more concrete view gives a full account of the logical/physical structure of the system we want to implement and can be used as a guide to the system implementation. Each refinement step is validated according to our efficiency preorder in order to make sure that each description behaves as dictated by the refined one while, in addition, its worst-case efficiency improves. During this stepwise refinement, we show how our preorder is able to remove useless non-deterministic sub-behaviours from the various descriptions.

We do not need generating extra models apart from the state transition system describing the dynamic behaviour of the system. Then, standard simulation-based relation techniques among system states can be exploited to verify whether or not a system description is more efficient than another. If \( P \) is not faster than \( Q \), i.e. \( P \not\preceq Q \), then there is a refusal trace of \( P \) that is not one of \( Q \). This is a witness of slow behaviour of \( P \); it is a diagnostic information that tells us why \( P \) is not faster. If \( P \) and \( Q \) are finite-state, inclusion of refusal traces can be checked automatically; a respective tool, FastAsy, has been developed for a Petri net setting [4], and we plan to adapt this for PAFAS. In case that \( P \) is not faster, FastAsy presents a respective refusal trace; this can be used to improve \( P \) – and in practice, it can also help to find errors that can occur when formalizing an intuitive idea as a PAFAS-process.

We are also investigating the possibility of deriving performance measures for the (worst-case) time needed by a given system to satisfy specific user requests.

References


