Evaluating the Efficiency of Asynchronous Systems with FASE*

F. Buti, M. Callisto, F. Corradini, M.R. Di Berardini
Dipartimento di Matematica e Informatica, Università di Camerino
{federico.buti, massimo.callisto, flavio.corradini,mariarita.diberardini}@unicam.it

W. Vogler
Institut für Informatik, Universität Augsburg
vogler@informatik.Zuni-Augsburg.kde

In this extended abstract we present a tool for evaluating the worst-case efficiency of asynchronous systems called FASE (**F**ast **A**synchronous **S**ystems **E**valuation). FASE is based on PAFAS algebra theory and we used it to relate three different implementations of the same bounded buffer already studied in preview works.

1 Introduction

PAFAS is a process algebra where basic actions are atomic and instantaneous but have associated a time bound interpreted as the maximal time delay for their execution. These upper time bounds can be used to evaluate efficiency, but they do not influence functionality (which actions are performed); so compared with CCS also PAFAS treats the full functionality of asynchronous systems. In [1], processes are compared via a variant of the testing approach developed by De Nicola and Hennessy [2]. Tests considered in [1] are test environments (as in [2]) 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 faster-than preorder relation over processes that is naturally an *efficiency preorder*. Furthermore, this efficiency preorder can be characterised as inclusion of a special kind of refusal traces which provide decidability of the testing preorder for finite state processes. In [3] it has been shown that the faster-than preorder of Corradini et al. [1] can equivalently be defined on the basis of a performance function that gives the worst-case time needed to satisfy any test environment (or user behaviour). Another main result in [3] shows that, whenever we adapt the timed testing scenario described above by considering only test environments that want n task to be performed as fast as possible, this performance function is *asymptotically linear*. This result provides us with a *quantitative* measure of systems performance, essentially a function (from natural numbers to natural numbers) - called *response performance function* - that evaluates how fast the system under consideration responds to requests from the environment. In this work we present FASE, a corresponding tool implementing the PAFAS theory, developed at the University of Camerino that supports us to automatically evaluate the worst-case performance of a PAFAS process. With FASE, we evaluated the efficiency of three different implementations of the same bounded buffer called Fifo (first-in-first-out queue), Pipe (sequence of cells connected end-to-end) and Buff (an array

*This work was supported by the PRIN Project 'Paco:Performability-Aware Computing: Logics, Models, and Languages'.

Submitted to:
QFM 2009

used in a circular fashion). The quantitative outcomes obtained were compared with the qualitative ones from [4], to show differences between the two preorders.

2 Process Algebra for Faster Asynchronous Systems

We use the following notation: $\mathbb{A}$ (ranged over by $a,b,c,\dots$) is an infinite set of basic actions that contains a special action ω reserved for observes (test processes) in the testing scenario to signal the success of a test. The action τ represent internal activity that is unobservable for other components. We define $\mathbb{A}_\tau = \mathbb{A} \cup \{\tau\}$; elements of $\mathbb{A}_\tau$ are ranged over by $\alpha,\beta,\cdots$. We assume that actions in $\mathbb{A}_\tau$ can let time 1 pass before their execution, i.e. 1 is their maximal delay. After that time they become *urgent*. The set of urgent actions is denoted by $\underline{\mathbb{A}}_\tau = \{\underline{a} \mid a \in \mathbb{A}\} \cup \{\underline{\tau}\}$ and is ranged over by $\underline{\alpha},\underline{\beta},\dots$. $\mathcal{X}$ (ranged over by $x,y,z,\dots$) is the set of process variables, used for recursive definitions. A *general relabelling function* is a function $\Phi : \mathbb{A}_\tau \to \mathbb{A}_\tau$ where the set $\{\alpha \in \mathbb{A}_\tau \mid \emptyset \neq \Phi^{-1}(\alpha) \neq \{\alpha\}\}$ is finite and $\Phi(\tau) = \tau$. As shown in [1], general relabelling functions subsume the classically distinguished operations relabelling and hiding.

The set $\mathbb{P}$ 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 $\alpha.()$, where $\alpha \in \mathbb{A}_\tau$) terms generated by the following grammar: $P ::= 0 \mid \gamma.P \mid P+P \mid P\|_A P \mid P[\Phi] \mid x \mid \mu x.P$, where γ is either α or $\underline{\alpha}$ for some $\alpha \in \mathbb{A}_\tau$, Φ 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. $\alpha.P$ and $\underline{\alpha}.P$ is the (action-) prefixing, known from CCS; $a.P$ can either perform a immediately, or can idle for time 1 and become $\underline{a}.P$. In the latter case, the idle-time has elapsed and a must either occur or be deactivated (in a choice-context) before time may pass further. Our processes are *patient*: as a stand-alone process, $\underline{a}.P$ has no reason to wait; but as a component in $\underline{a}.P\|_{\{a\}}a.Q$, it has to wait for synchronisation on a 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 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 synchronise on all actions from A. This discipline of synchronisation is inspired by TCSP.

The temporal behaviour of PAFAS processes is given by means of the so-called *refusal traces*. Intuitively, a refusal trace records along a computation which actions a process P can perform ($P \xrightarrow{\alpha}_r P'$, $\alpha \in \mathbb{A}_\tau$) and which actions P can refuse to perform ($P \xrightarrow{X}_r P'$, $X \subseteq \mathbb{A}$).[1] A transition like $P \xrightarrow{X}_r P'$ is a *conditional time step*. The actions listed in X are not urgent; thus P is justified in not performing them, and performing a conditional time step instead. Since other actions might be urgent, P might actually be unable to refuse any possible action (e.g. $\underline{a}.P$ can never refuse a). Nevertheless, as a components of a larger system, P can refuse some of its urgent actions due to synchronisation with the environment. For example, as a component of $\underline{a}.P\|_{\{a\}}a.Q$, $\underline{a}.P$ can refuse the action a since it has to synchronize on this action with the environment Q and the latter can refuse it. P can make a *full time step* if $P \xrightarrow{\mathbb{A}}_r P'$. In such a case we also write that $P \xrightarrow{1} P'$. A *discrete trace* is any sequence $v \in (\mathbb{A}_\tau \cup \{1\})^*$ that P can perform. Finally, $\mathsf{DL}(P)$ and $\mathsf{RT}(P)$ are the sets of discrete traces and refusal traces (resp.) of P.

[1] We omit here the (almost standard) SOS-rules defining the transition relations $\xrightarrow{\alpha}$ and $\xrightarrow{X}_r$ (see [1] for further details).

2.1 Qualitative and Quantitative performance evaluation

The efficiency preorder defined in [1] is a timed variation of De Nicola/Hennessy testing preorder. Unlike [2], the timed tests considered in [1] are pairs (O,D) where O is a test environment (or user behaviour, i.e. a process that contains ω) and $D \in \mathbb{N}_0$ is an upper time bound. A testable process P (i.e. a process that can never perform ω) satisfies a timed test (O,D) if each $v \in \mathsf{DL}(P \|_{\mathbb{A}\setminus\omega} O)$ with $\zeta(v) > D$ contains ω (we use $\zeta(v)$ to denote the duration, i.e. number of 1's in v). P is faster than Q, written $P \sqsupseteq Q$, if P satisfies *all timed tests* that Q satisfies. Moreover, $P \sqsupseteq Q$ iff $\mathsf{RT}(P) \subseteq \mathsf{RT}(Q)$ (for more details see [1]) which provides a decidability result for finite state processes.

This efficiency preorder is qualitative in the sense that a timed test is either satisfied or not, and that a process is more efficient of another or not. In order to bring to light its quantitative nature, in [3] the authors provides a new formulation of such a preorder that is based on a *performance function* p. Basically, if P is a testable process and O is a user behaviour, the function $p(P,O)$ gives the worst-case time P needs to satisfy O. In [3] it has also been proven that $P \sqsupseteq Q$ iff $p(P,O) \leq p(Q,O)$ for all test process O. The performance function p (as the preorder $\sqsupseteq$) contrasts processes w.r.t. any possible test environments. In some cases this might be too demanding and one can make some reasonable assumption about the user behaviours. In particular, one could be interested in users that have a number of requests (made via an *in*-action) that they want to be answered (via an *out*-action) as fast as possible. This is the class of users $\mathscr{U} = \{U_n \,|\, n \geq 1\}$ where $U_1 \equiv \underline{in}.\underline{out}.\underline{\omega}$ and $U_n = U_{n-1} \parallel_{\{\omega\}} \underline{in}.\underline{out}.\underline{\omega}$ (for any $n > 1$). Given these users, we can define the *response performance* rp_P as a function from $\mathbb{N}$ to $\mathbb{N}_0$ such that $rp_P(n) = p(P,U_n)$ (here n is the size, i.e. the number of requests of the user). In what follows, we briefly describe the approach followed in [3] in order to calculate the response performance of any process that can reasonably serves users in $\mathscr{U}$. This processes are called *response processes*[2] in [3].

We first observe that in order to determine $rp_P(n)$ we have to consider all $v \in \mathsf{DL}(P \parallel U_u)$ that do not contain ω, count the number of 1's that such traces contain, and finally take the supremum of the numbers so obtained. Now, traces in $\mathsf{DL}(P \parallel U_u)$ are just paths in $\mathsf{RTS}(P \parallel U_u)$ that only contain full time steps. Moreover, for each path in $\mathsf{RTS}(P \parallel U_u)$ there exists a corresponding path in $\mathsf{rRTS}(P)$[3] with the same number of conditional time steps. Thus, to calculate $rp_P(n)$ it will suffice to consider path in $\mathsf{rRTS}(P)$. A first result in [3] states that the response performance of a response process P is the supremum of the number of time steps taken over all paths in $\mathsf{rRTS}(P)$ with enough *in*'s and *out*'s to satisfy the user U_n (we call such traces *n-critical paths*). At this point a key observation is that, when the number n of requests is large compared to the number of processes in $\mathsf{rRTS}(P)$, a n-critical path with many time steps must contain cycles. Thus, it turns out to be essential to find the worst cycles. In [3] these worst-cycles are distinguished to be either catastrophic or bad cycles.

A cycle in $\mathsf{rRTS}(P)$ is said to be catastrophic if it has a positive number of time steps but no *in*'s and no *out*'s. Intuitively, if $\mathsf{rRTS}(P)$ contains a catastrophic cycle then there is at least an admissible path in $\mathsf{rRTS}(P)$ with arbitrarily many time steps and, hence, there is an n with $rp_P(n) = \infty$. If P is a response process without catastrophic cycles, rp_P is asymptotically linear and there is an $a \in \mathbb{R}$ such that $rp_P(n) = an + \Theta(1)$ (see Theorem 16 in [3]). The *asymptotic factor* a of $rp_P(n)$ is determined by considering cycles reached from P by a path where all time steps are full and which themselves contain only time steps that are full; let the *average performance* of such a cycle be the number of its full time

[2] A processes is a response process if it can only perform *in*'s and *out*'s as visible actions and can never produce more responses than requests.

[3] A reduced version of $\mathsf{RTS}(P)$. See [3] for more details.

steps divided by the number of its *in*'s. We call a cycle *bad* if it is a cycle of maximal average performance in $\mathsf{rRTS}(P)$. Finally, the asymptotic factor of P is the average performance of a bad cycle.

3 The tool Fase

The tool FASE has been developed to automatically evaluate the worst-case efficiency of asynchronous systems. It is written in Java and consists of two main components loosely coupled. The first one is a parser unit that takes as input a sequence of characters that represents a PAFAS process and builds its RTS. The second component implements all the technical stuffs introduced in the previous section.
In [3], the problem of finding catastrophic cycles in $\mathsf{rRTS}(P)$ has been reduced to that of finding the shortest paths in an arc-labelled graph G_P obtained as a weighted modification of $\mathsf{rRTS}(P)$. All the edges (and all nodes not reachable any more) labelled with an *in*- and with an *out*-action are removed from $\mathsf{rRTS}(P)$ and a weight -1 (0) is assigned to edges that corresponds to a time step (a τ-action, respectively). On this graph so obtained, the Floyd-Warshall algorithm is used to compute all shortest paths. We are only interested in cycles, i.e. paths from a given node m of G_P to itself. These correspond to values on the the diagonal of shortest path matrix $D \in \mathbb{N}^{N\times N}$ computed by Floyd-Warshall algorithm (N in the number of nodes in G_P). Moreover, if $D[m,m] < 0$ the cycle contains at least a time step, and hence, it is catastrophic. As well-known, Floyd-Warshall algorithm computes the matrix D is $O(N^3)$.

The solution adopted in FASE takes advantages of the the so-called finding problem of *Strongly Connected Components* (*SCC*) [5]; they are subgraphs of $\mathsf{rRTS}(P)$ that are strongly connected and maximal. It is sufficient to check in the building process of the SCCs (where we do not take in account *in* and *out* actions), if there are some time steps to recognize a catastrophic cycle. Indeed, if Q is one of such components and there is a time step between two state processes R and S in Q, then the cycle in Q that contains R and S is catastrophic by definition. Since the algorithm is essentially a depth-first visit of the graph $\mathsf{rRTS}(P)$, its complexity is $\Theta(N+M) = O(N^2)$ where N and M are the numbers of nodes and the number of edges (respectively) in G_P. To find out bad cycles, FASE implements the algorithms originally proposed in [3] which has a running time of $O(N^3)$ where N is the size of $\mathsf{rRTS}(P)$. However, we are investigating some different solutions to improve this running time due to the common *state explosion problem* which represent a big issue when it comes to analyse real large complex systems. For the same reason we are trying to define some new method to fix the n-critical paths value as for now it needs an outstanding amount of computational resource.

4 Main results and concluding remarks

We have tested FASE over a wide range of examples. In particular, we have used FASE to evaluate the worst-case efficiency of three different implementations of a bounded buffer of capacity $N+2$ where N is a fixed natural number. The worst-case behaviour of the same implementations have already been studied in [4] and we want to investigate if the results stated in [4] still hold in our qualitative setting. The first implementation of the buffer Fifo is a bounded-length first-in-first-out queue; it has no overhead in terms of internal actions and it is purely sequential. Process Pipe implements the buffer as the concatenation of $N+2$ cells, where each cell is an I/O device able to store at most one value. The cells are connected end-to-end, i.e. the output of each cell becomes the input of the next one. Finally, Buff uses N cells as

a storage that interacts with a centralised buffer controller. The buffer controller manages the cells in the storage in a circular fashion and also retains the oldest undelivered value and outputs it whenever possible.

By using FASE we was able to prove that the three buffer implementations we have considered do not have catastrophic cycles. Moreover, experimental results showed that $rp_{\mathsf{Fifo}}(n) = 2n$, $rp_{\mathsf{Pipe}} = 2n+N+1$ and $rp_{\mathsf{Buff}}(n) = 4n$. Thus we can conclude that, Fifo is more efficient (from a quantitative point of view) than both Pipe and Buff; moreover, Buff is more efficient than Pipe iff $n \leq \lfloor \frac{N+1}{2} \rfloor$.

These results are quite different from those presented in [4] where the efficiency of the same buffer implementations has been compared by means of the efficiency preorder defined in [1]. In [4] it is stated that Fifo and Pipe but also Buff and Pipe are unrelated according to the worst-case efficiency preorder (unrelated means that the former process is not more efficient of the second one and vice versa). In [4] the authors provide good reasons for these results and also prove that Fifo is more efficient than Buff but not vice versa. Our conviction is that here (as in [3]) we contrast processes w.r.t. to a specific class of user behaviours while the testing scenario we use in [4] contrasts process w.r.t. any possible user. To prove if our intuition is correct, we are working on the definition (and characterisation) of a slight variation of the faster-than preorder as it is given in [1]. Basically, this new preorder should allows us to contrast processes only w.r.t. user behaviours in $\mathscr{U}$. Moreover, it still remains to investigate in which extent the approach described in [3] can be generalized to other possible scenarios and to a different (maybe larger) class of user behaviours. For what concerns FASE, our aim is to make it a suitable solution to analyse a larger set of systems. We worked hard on the parser unit to make it as much independent and flexible as possible to respond to the theoretical progress. The performance module needs more attention since it implements all the theories introduced above. The first result that we obtained is to improve the catastrophic cycles detection. This is very useful since ensuring their absence is the basis for any further performance analysis. Anyway, a number of issues are still open. First of all, since we work with LTSs, the problem of *state space explosion* can be often phased. As well-known, different solution based on *on the fly techniques* can be exploited in order to reduce this problem and, in the next future, we intend to investigate how these techniques could help us to improve FASE applicability. We are also working on possible different solutions that would pick up implementations of the algorithms we use to find both bad cycles and n-critical paths.

References

[1] Corradini, F., Vogler, W., Jenner, L.: Comparing the worst-case efficiency of asynchronous systems with PAFAS. Acta Informatica **38**(11) (2002) 735–792

[2] De Nicola, R., Hennessy, M.: Testing equivalences for processes. TCS **34** (1984) 83–133

[3] Corradini, F., Vogler, W.: Measuring the performance of asynchronous systems with pafas. Theor. Comput. Sci. **335**(2-3) (2005) 187–213

[4] Corradini, F., Berardini, M.D., Vogler, W.: Pafas at work: Comparing the worst-case efficiency of three buffer implementations. Asia-Pacific Conference on Quality Software **0** (2001) 0231

[5] A. V. Aho, J. E. Hopcroft, J.D.U.: Data Structures and Algorithms. Addison-Wesley (1983)