The pursuit of buffer tolerance

A.W. Roscoe*
Oxford University Computing Laboratory

May 5, 2005

Abstract

A system is buffer tolerant when buffers (namely, queues that may be of arbitrary or varying length) may be introduced onto some or all of its channels without introducing errors. We give formal definitions of several types of buffer tolerance within the context of CSP and its models, prove a number of results about it and discuss when these might be useful. Most of our results apply to tree networks and to ones where the processes involved have a property such as being *functional* or *confluent*. We demonstrate the close connection of these last two properties by showing that they can both be characterised as appropriate sorts of buffer tolerance.

1 Introduction

The term *buffer tolerance* refers to systems into which buffers may be introduced onto their channels without introducing any extra errors. It is important in verifying real concurrent systems because it is likely to be much more tractable to verify a system model without buffers, mainly because of the reduction in state-space that this brings. This is certainly the case on with the CSP refinement checker FDR [**?**].

The term *buffer* here is nearly synonymous with *queue*, but there is no implication that it is unbounded. A buffer may be unbounded, but equally it

*`bill.roscoe@comlab.ox.ac.uk`

may refuse further input whenever it is nonempty. A nonempty buffer never refuses output. Buffers have been studied extensively in the CSP literature, for example [18].

Buffer tolerance is one of a number of approaches which can be used to bring systems with buffered (and perhaps fully asynchronous) channels within the realm of model checkers. Others typically abstract the buffers on channels in such a way that they become finite state, perhaps through being modelled as regular languages and related forms of BDDs [1, 2, 5]. Buffer tolerance is just one aspect of the *Parameterised Verification Problem.* Where possible the prospect of eliminating buffers entirely from a system is attractive since it has the advantages of simplicity: both conceptual and in terms of expected state-space when we come to model-check.

Such techniques are important because buffered and asynchronous channels are found in many applications, frequently making them impossible to model-check directly.

Buffer tolerance is a complex subject because of the number of ways it can arise and because defining it is not nearly so clear cut as, for example, in deadlock freedom. The author has visited this topic twice before [6, 18] but many questions remained unanswered.

With the advantage of the specification tools and insight developed in [20, 19] it has been possible to derive some significantly stronger results than hitherto. This paper is in some ways modelled on the author's similarly-named paper on deadlock from about 20 years ago [22]. It is in two parts. The first one sets out the basics and – like the first part of [22] – looks at networks with no cyclic dependencies (though we do give some general results that go beyond these). The second looks mainly at a particular class of processes: ones like buffers where the stream(s) on the output channels depend monotonically on the input stream(s). We call these *functional.* These are related to *data flow* systems: they have strong buffer tolerance properties and we have some strong results about characterising and using them.

The rest of this paper is organised as follows: in the next section we look at the problem of specifying buffer tolerance and we derive some basic results. In doing so we experiment with a number of conditions on processes such as variations on the concept of *receptiveness* which regulate when they can refuse further input. We end up with a number of variants on buffer tolerance such as *weak* and *strong.* In the following section we prove that all chains of processes in which data flows in one direction have a form of buffer tolerance, and extend this to classes of tree networks as well at looking at more general trees. In the final section of Part 1 we examine local criteria for buffer tolerance and seek finitary methods of discharging the preconditions of our proof rules.

In Part 2 we examine the sort of specification which we can reasonably hope to prove of weakly buffer tolerant systems. This leads to the ideas of data-flow computation, and derive the natural class of trace specifications for these processes. In the following section we define *functional agents* as a subclass of these and show how the latter can be extended to failures/divergences specifications. We then show how networks built up from functional agents are always buffer tolerant with respect to traces, and examine the extent to which this idea can be extended to failures. In the next section, we show how the concept of confluence [15] can be used to generalise the idea of receptiveness and how it gives rise to interesting buffer tolerance results in its own right. Finally, we derive a number of characterisations of confluence and of functional agents, some of which are both practical (being finitary) and surprising.

A by-product of this work is a new technique for deciding whether or not a given finite-state process has any given functional behaviour, such as being a standard CSP buffer.

There is an appendix summarising CSP and other notation.

This paper is based on the process algebra CSP and its models. Many of our ideas and concepts would transfer (in general terms, at least) to other settings. The author suspects, however, that it might be difficult to transfer many of our failures/divergences model results to more discriminating equivalences such as bisimulation.

Part 1: The Basics

2 Specifying buffer tolerance

Buffer tolerance only makes sense of channels that have a definite direction, rather than having components which convey information in both directions. The following assumptions apply only to channels that are candidates for buffer tolerance and global input and output channels of the same sort. In other words we imagine that there may be other synchronisations in our network. (Most of the results, and all the examples, in this paper relate to cases where all channels are potentially buffered, however.)

While buffer tolerance could make sense for broadcast communication we will not consider that or any other communication involving more than two partners. It is usually convenient to assume that the communications on the relevant channel are hidden as in the CSP chaining or piping operator $\gg$.

Except when explicitly restricting analysis to the traces model of CSP, we will assume that all component processes we consider are divergence free. We will usually assume properties of these processes that will exclude divergence in the complete network.

Let the sets of input and output channels of process P be denoted $\mathit{Inchans}(P)$ and $\mathit{Outchans}(P)$ respectively[1].

We also make two assumptions which reinforce the idea that channels have direction.

DEFINITION 2.1 *We say a process P has the* no-selective-input *property on input channel c if the following hold:*

(i) if, after some trace s, P has the refusal $X \cup \{c.a\}$, then $(s, X \cup \{| c |\})$ is in $\mathit{failures}(P)$.

(ii) if $s\hat{\ }\langle c.a\rangle$ is a trace of P, then so is $s\hat{\ }\langle c.b\rangle$ for every member b of the type of c.

These correspond to the idea that when P offers any communication on c then it offers the environment a complete choice.

DEFINITION 2.2 *We say P is* output decisive *if, whenever $(s, X) \in \mathit{failures}(P)$ then there is $Y \supseteq X$ such that $(s, Y) \in \mathit{failures}(P)$ and, for every output channel c, $\{| c |\} \setminus Y$ contains at most one element. This is similar to, but a little stronger than, the statement that a process can make an output on c then it can offer no choice of values for the input process to select from.*

We say it is strongly output decisive *if, in addition, whenever $s\hat{\ }\langle c.x\rangle \in \mathit{traces}(P)$ for $c.x$ an output action, then there is X such that $(s, X) \in \mathit{failures}(P)$ such that $\{| c |\} \subseteq X \cup \{c.x\}$.*

[1]As with the process alphabets of [7], it is dangerous to attempt to calculate these from the semantics of a process. Therefore it is assumed that these are defined with the process.

Any process which has no-selective-input buffers connected to its input and output channels is *(in toto)* strongly output decisive and has no-selective-input on all its input channels. We will assume that these properties hold for each sequential component process and channel we consider.

Neither (strong) output decisiveness nor no-selective-input is a simple behavioural property. It is, however, relatively straightforward to formulate refinement checks for them in the style of [19, 20].[2] No-selective-input and output decisiveness are both compositional under parallel composition that connects processes by whole channels and hides the resulting internal actions. Strong output decisiveness is not, as witnessed by the following example.

---

[2]No-selective-input can be decided by FDR refinement checks as follows. Here we show how to do it for all input channels: this can easily be adapted for one or several. Let OPC be the nondeterministic process with alphabet I – the events from the input channels of P – which can perform any trace of these events and which can refuse any set which has no whole channel as a subset. P satisfies part (i) of the condition if and only if $OPC \underset{I}{\parallel} P$ only refuses a whole input channel when P does. This is equivalent to the statement that $F(P) \sqsubseteq F(OPC \underset{I}{\parallel} P)$ where

$$F(Q) = ((Q[\![^{i,\, e.c(i)}/_{i,\, i} \mid i \in I]\!]) \underset{I}{\parallel} Chaos_I) \underset{\Sigma}{\parallel} OneE$$

Here, for each input event i, $c(i)$ is its channel name, e is chosen so that each $e.c$ is an event not in the alphabet of P, and $OneE$ is a process which forces the whole construct to $STOP$ after the first action labelled e, but before that permits any action.

For part (ii) of the condition, we suppose that there is a shadow copy c' of each input channel. Let One' be the process similar to $OneE$ that stops after its first c' event for any c, and let

$$G(P) = P[\![^{c,\, c'}/_{c,\, c}]\!] \underset{\Sigma}{\parallel} One'$$

be the process that can at any time perform $c'.a$ where P performs $c.a$ but which then has to stop. Since we want to make sure that P can perform every input on each input channel c when it performs one, we must test

$$G(P) \sqsubseteq_T G(P)[\![R]\!]$$

where R is the relation $\{(c.x, c.y) \mid c \in Inchans(P), x, y \in T(c)\}$. This tests precisely what we need, and is equivalent to part (ii).

The requirements of (strong) output decisiveness can be captured using similar constructions.

Example 2.1 Consider the process

$$P = out!empty \rightarrow STOP \,\Box\, in?x \rightarrow out!full \rightarrow STOP$$

P is clearly strongly output decisive, as is $COPY$. $COPY \gg P$ is not strongly output decisive because it has the trace $\langle in.x, out.empty\rangle$ but no suitable refusal. This is because after $COPY$ has made an input but before P has received it, P can output *empty*, but cannot come into a stable state where it can do so.

Thus *strong* output decisiveness is not generally true of processes which can make an output in an unstable state which is different from the values that can be output in following stable states. ∎

Buffers have been widely studied in CSP since its invention, since they are practically important in their own right, provide a model on which to build richer specifications, and have an interesting theory (see [18], for example). The standard concept of a buffer in CSP is a process with one input channel and one output channel (*left* and *right*, say) both of which have the same type. It satisfies the following specification:

(i) All a buffer does is input on *left* and output on *right*. It correctly copies all its inputs to its output channel, without loss or reordering.

(ii) Whenever it is empty (i.e., it has output everything it has input) then it must accept any input.

(iii) Whenever it is non-empty, then it cannot refuse to output.

This can easily be translated into a specification in terms of failures and divergences. Since a diverging process is not responding to its environment; whether an input or output is expected of it, it is clear that we cannot allow a buffer to diverge. We therefore stipulate that, for any buffer B, $divergences(B) = \{\}$. The rest of the specification is then

(i) $s \in traces(B) \Rightarrow s \in (left.T \cup right.T)^* \wedge s \downarrow right \leq s \downarrow left$

(ii) $(s, X) \in failures(B) \wedge s \downarrow right = s \downarrow left \Rightarrow X \cap left.T = \{\}$

(iii) $(s, X) \in failures(B) \wedge s \downarrow right < s \downarrow left \Rightarrow right.T \not\subseteq X$

These conditions simply translate the corresponding English clauses into mathematics.

In buffer tolerance we are interested in what happens if processes such as these are placed on some or all of the channels of a system. In fact we will want to be slightly more general than this and have channels where there may sometimes be buffering and sometimes not. By "arbitrary buffering" below we will mean that the given channel is replaced by this type of behaviour. We will give concrete implementations of it shortly.

Buffer tolerance is a property of a set of channels – maybe a single channel or all internal channels – in the context of a network. More than that, it can only really be judged relative to a given specification one wants to prove of that network. To see the latter, observe that the channel in $COPY \gg COPY$ ($COPY$ being the standard CSP one-place buffer process) is buffer tolerant (on its only channel) relative to the specification of being a buffer, but not relative to it being a two-place buffer! Of course a channel may be tolerant of some buffers (perhaps those with some bound on capacity) but not others.

This just illustrates that while the introduction of buffering may not change the basic intuitive behaviour of a system, it is quite likely to alter the phasing of externally visible events. Of course it is possible to avoid any external change of this sort by using suitable flow-control such as acknowledgement signals in a network (as used, for example, in the alternating bit protocol), but we do not want to be restricted to that sort of network.

We therefore define a number of variants on buffer tolerance. Before doing so we introduce some notation: if N is any network then $\mathit{buff}(N)$ means N with arbitrary buffering added to all external channels of N (i.e. $\mathit{Inchans}(N) \cup \mathit{Outchans}(N)$). $\mathit{inbuff}(N)$ means N with arbitrary buffering applied to the members of $\mathit{Inchans}(P)$ and $\mathit{outbuff}(N)$ means N with arbitrary buffering applied to members of $\mathit{Outchans}(N)$. Assume we are examining a set C of channels within a network process N for buffer tolerance. $N^{\diamond C}$ will mean N with arbitrary buffering added to all members of C, if C is all internal channels we write $N^{\diamond}$, and if it is all internal and external channels we write $N^{\diamond\diamond}$ ($= \mathit{buff}(N^{\diamond})$). See below for formal definitions, and more notation, on the topic of "arbitrary buffering". In the following we assume all internal communication in N is hidden, and we are investigating a set C of N's internal (and sometimes external) channels.

Definition 2.3 *The following definitions can each apply to any one of the* *standard CSP models. If, in any instance, no specific model is quoted, it will default to the failures/divergences one – equivalent to the stable failures one for divergence-free systems.*

- *The network is* strongly buffer tolerant *with respect to C if the introduction of an arbitrary buffer does not change the overall semantics of the system. Namely, $N = N^{\diamond C}$.*

- *It is* leftwards buffer tolerant *if $N^{\diamond C}$ refines inbuff(N), and* rightwards buffer tolerant *if $N^{\diamond C}$ refines outbuff(N).*

- *It is* weakly buffer tolerant *if $N^{\diamond C}$ refines buff(N).*

- *It is* strongly buffer tolerant relative *to a specification S if adding an arbitrary buffer onto C cannot make N fail to satisfy S, namely $N \sqsupseteq S \Rightarrow N^{\diamond C} \sqsupseteq S$.*

- *It is* weakly *buffer tolerant relative to a specification S if $N \sqsupseteq S \Rightarrow N^{\diamond C} \sqsupseteq$ buff(S). It is* leftward *or* rightward *buffer tolerant relative to S if, after adding the buffers, it refines respectively inbuff(S) or outbuff(S).*

- *It is* locally *buffer tolerant if, for each P in N, the set of states P can reach in $N^{\diamond C}$ is the same that it can reach in N.*

Notice that makes sense to say that a single process (as opposed to a network) is leftward or rightward buffer tolerant. These respectively say that $\mathit{inbuff}(P) \sqsubseteq \mathit{outbuff}(()P)$ or $\mathit{outbuff}(P) \sqsubseteq \mathit{inbuff}(()P)$, or, equivalently, $\mathit{inbuff}(P) = \mathit{buff}(P)$ or $\mathit{outbuff}(P) = \mathit{buff}(P)$.

Note that N is leftward (rightward, weakly) buffer tolerant with respect to all channels if and only if $N^{\diamond\diamond}$ is equivalent to $\mathit{inbuff}(N)$ ($\mathit{outbuff}(N)$, $\mathit{buff}(P)$).

Local buffer tolerance is not formally weaker or stronger than the others, since it is actually judged in a very different way (in operational rather than abstract semantics). Its name comes from the fact that any failure is judged in terms of a local observation of one of the constituent processes in N rather than in terms of overall network behaviour. A network that fails it is unlikely to have useful buffer tolerance properties, and one that does will do so because its components' behaviours outside the range exercised in N are like those behaviours which are. There is then no hope of simulating buffered network behaviour by unbuffered without relying on semantic properties of the component processes. As we shall see later (Section 4), it is generally straightforward to prove in networks with no cycles amongst the channels, so we will not discuss it much till then.

The left, right and weak definitions, and their "relative" counterparts, are naturally linked by the following lemma. The corresponding result for the strong definitions is obvious.

LEMMA 2.1 *If N is weakly buffer tolerant (or to left or right) in respect of channels C, then it is weakly buffer tolerant for C with respect to every specification S (or to left or right if appropriate).*

PROOF Suppose $N \sqsupseteq S$. Then $\mathit{buff}(N) \sqsupseteq \mathit{buff}(S)$ by monotonicity. (We are implicitly assuming that the direction of every channel in N is the same as it has in S.) Hence, by weak buffer tolerance and transitivity, we have $N^{\diamond C} \sqsupseteq \mathit{buff}(S)$. This is what we required for the symmetric case. The left and right cases are very similar. ∎

We now give explicit CSP definitions for $\mathit{buff}(P)$ and related constructs. These are rather subtle: simply linking P to an appropriately named and oriented copy of the most nondeterministic (empty) buffer $BUFF_{\langle\rangle}$ does not achieve what we want, since it adds a buffer of length at least one on each channel.[3] Nor (even for a single channel) does the construction $P \sqcap (P \gg BUFF_{\langle\rangle})$, since once some buffering is observed on a trace, at least a one-place buffer is always available: this construct on either channel of $COPY$ would guarantee that as soon as two items have been in it, inputting a second item is never refused.

The answer (like so many other difficult representation problems in CSP[4]) can be found in double renaming: mapping some events to two alternatives each. Suppose we want to add the most nondeterministic zero-or-more place buffer to $c \in \mathit{OutChans}(P)$. Let c_1 and c_2 be two new channel names with the same type as c. Define a new process ZB_s for each finite sequence s from

[3] In fact the corresponding construction in the traces model, namely adding an unbounded buffer onto each channel *does work* since bounds the only bounds can be detected by the presence of a behaviour is via refusals.

[4] We have already used it when giving a refinement representation of no-selective-input.

the type of c as follows:

$$\begin{array}{rcl} ZB_{\langle\rangle} & = & (c_1?x \to ZB_{\langle\rangle}) \\ & & \sqcap\, (c_2?x \to ZB_{\langle x\rangle}) \\[1ex] ZB_{s\hat{}\langle y\rangle} & = & (c!y \to ZB_s) \\ & & \Box\, (STOP \sqcap c_2?x \to ZB_{\langle x\rangle\hat{}s\hat{}\langle y\rangle}) \end{array}$$

(ZB stands for *zero buffer.*) ZB_s behaves exactly like $BUFF_s$ (the most nondeterministic buffer initially containing s) from c_2 to c except that when empty it can also communicate on c_1. If c is an output channel of P, this enables us to create the process

$$(((P[\![^{c_1, c_2}/c, c]\!]) \underset{\{|c_1,c_2|\}}{\|} ZB_{\langle\rangle}) \setminus \{|\ c_2\ |\})[\![^c/c_1]\!]$$

in which, in addition to the possibility of being buffered, communications on c can, via the c_1 option in $ZB_{\langle\rangle}$, be transmitted directly to the environment. A symmetric construction works for input channels. We will denote the addition of arbitrary buffering to an output channel by $P\diamond c>$, and $\diamond c>P$ for an input channel. Where the channel name is obvious we will just write $P\diamond>$ or $\diamond> P$ as appropriate.

Applying this construction (appropriately oriented) to each of N's channels gives us *buff*(N), and to input or output channels gives us *inbuff*(P) and *outbuff*(P).

To insert arbitrary buffering on an internal channel c of a network, simply apply the above construction (on c) to the process outputting on c before joining it to the inputter. The link parallel operator (see [18], for example) between two processes can be extended to incorporate arbitrary buffering on certain channels by replacing the $a \leftrightarrow b$ of an unbuffered link by $a \diamond\!\!\longrightarrow b$ for a rightwards buffer or $a \longleftarrow\!\!\diamond b$ for a leftwards one. So

$$P[a \leftrightarrow a, outp \diamond\!\!\longrightarrow inq, inp \longleftarrow\!\!\diamond outq]Q$$

links P and Q by three channels: one synchronised, and one buffered for each direction.

Where the single channel connecting P is implicit as in $P \gg Q$, we will denote the addition of arbitrary buffering to it by $P >\!\!\diamond\!\!> Q$.

It is important to realise that, because $ZB_{\langle\rangle}$ has the option always to force synchronisation between its two sides, that all of the above constructions produce a result which is refined by the corresponding system without buffering. This property was used in the proof of Lemma 2.1. For example

$$N \sqsupseteq \mathit{inbuff}(N) \sqsupseteq \mathit{buff}(N)$$

(where $\mathit{inbuff}(N)$ can be replaced by $\mathit{outbuff}(N)$).

ZB_s is, of course an infinite-state process which means it cannot be used directly on FDR. In Chapter 5 of [18] both upper and lower (weak) finite-state approximations to the most nondeterministic ordinary buffer are quoted: these can easily be adapted to ZB, as can the methods described there for using them.

It is sometimes useful to have analogous notation for the addition of an unbounded deterministic buffer on one or more channels. Replacing '$\diamond$' by '∞' in operators such as $\infty\!> P$ will have the obvious definitions, and the corresponding version for the operators applying to all, input, and all output channels will be $\mathit{buff}^{\infty}(P)$, $\mathit{inbuff}^{\infty}(P)$ and $\mathit{outbuff}^{\infty}(P)$.

It is worth noting that a network can have any one of our flavours of buffer tolerance with respect to two channels c and d individually without being buffer tolerant with respect to $\{c, d\}$. For example c and d might be on alternative routes for data to pass from P to Q. If buffering is placed on one of them, it can still be possible for the delay in the data reaching Q to be bounded, but not if infinite buffering is placed on them both. If there were a further channel between P and Q, this might mean that an entirely different state of one or other could be reached with two buffers that could not be reached with either: that channel might check in some way that the delay over the other two was bounded. In other words it might not be locally buffer tolerant.

Before proceeding further about buffer tolerance, we will state some definitions which will be useful to us. In the following, we assume that the alphabet of the process P is partitioned into inputs I and outputs O.

DEFINITION 2.4 *We say P is* accepting *if and only if it is deadlock free and furthermore, if $(s, X) \in \mathit{failures}(P)$ and $O \subseteq X$, then $I \cap X = \{\}$.*

It is receptive *if it never refuses any input, namely $(s, X) \in \mathit{failures}(P) \Rightarrow X \cap I = \{\}$.*

"Accepting" simply says that the process either has to offer an output or accept any input. This is obviously related to many familiar specifications such as that of a buffer. "Receptive" is a condition familiar in the world of dataflow computation, e.g. [8].

Note that a process with only one input channel and which has the no-selective-input-property is deadlock free if any only if it is accepting.

"Receptive" says that the process has an infinite appetite for inputs which is unaffected by what does or doesn't happen on its output channels. The only receptive buffer is the infinite deterministic one $B^{\infty}_{\langle\rangle}$.

Definition 2.5 *Suppose the process P cannot perform an infinite number of inputs without performing an output (namely, $P \setminus \{| \, in \, |\}$ is divergence free); call this is* finite input property*, or FIP.*

The finite output property *(FOP) is the dual of this: P cannot perform an infinite number of outputs without an input, equivalent to $P \setminus \{| \, out \, |\}$ being divergence free.*

Note that the finite input property and being receptive are mutually inconsistent.

The following lemma (where (c) and (d) are results that have been known in the CSP community for many years) shows how two of these properties are inherited by acyclic networks from their components.

Lemma 2.2 *Suppose N is a network in which all internal communications are along directed, hidden, channels, and where there is no cycle in the directed graph of channels. Then*

(a) If all processes in P are accepting, and N is divergence free, then N is accepting.

(b) If all processes in P are accepting, N is divergence free, and furthermore every path from an input channel of N to an output channel contains a receptive node, then N is receptive.

(c) If all processes in P have the finite output property, then N is divergence free and has the finite output property.

(d) If all processes in P have the finite input property, then N is divergence free and has the finite input property.

PROOF We first prove (a), by induction on the size of N. The result is trivially true when this is one, so suppose it is true for all smaller networks. Since the graph of channels is acyclic we may choose a process P all of whose inputs are external; let the remainder of the network be N'. By induction N' is a accepting process. If N were not there would be a stable state it could reach where all external outputs are refused, and some input is refused. (Stability here comes from our assumption of divergence freedom.) Since none of N''s outputs are connected to P we can guarantee that they are all refused by N'. It follows that N' is, in our state, accepting all inputs. Since the network is stable P must be refusing all outputs (both those which are external to N and those connected to N') and therefore refusing no input. Since every input of N is one of P or N' it follows that no input of N is refused, contradicting our assumption. The result follows.

For (b), suppose that N refuses an input action a_0. Then the node which is refusing this must be able to output an action a_1 (as it is accepting). As the state is stable (as any refusing state is), the inputter of a_1 must be refusing that, and so is able to output some a_2, and so on. Since N is acyclic, this sequence would eventually reach an output of N, but cannot do so because the path eventually contains a receptive node. Since the latter cannot refuse to input the relevant a_i, we conclude that N is receptive.

(c) and (d) have symmetric proofs: we will concentrate on (c). Again we work by induction on the size of N and again the base case is trivial. Decomposing N into $N' \cup \{P\}$ as before, we see that any behaviour of N with a finite number of inputs can only include a finite number of outputs from P (as all of P's inputs are external and P has the finite output property). Since by induction N' has the property also, and all its inputs are either external or from the finite number of outputs P produces, N' can only produce finitely many outputs. As all outputs of N are outputs of P or N' it follows that N has the finite output property.

That N is divergence free follows easily from the fact that any divergence introduced by combining P and N' would require an infinite unbroken trace of hidden internal actions – all outputs of P – which is impossible because P has the finite output property. ∎

LEMMA 2.3 *(a) If P is accepting, or has the finite output property, then $\diamond c\!>P$ and $P\diamond d\!>$ have the same property for all appropriately oriented channels.*

(b) *If P is divergence free then so is $\diamond c\!>P$; if P has FOP, then $P\diamond d\!>$ is divergence free.*

PROOF For *accepting* in part (a), observe that neither $\diamond c\!> P$ nor $P\diamond d\!>$ can stably refuse an output when P does not; and neither can stably refuse any input when P does not. Since P cannot do both of these two things simultaneously, neither can either buffered process.

The result for the *finite output property* follows easily from the fact that the buffering also has this property.

Note that the result is not true for the finite input property since the arbitrary buffer can absorb an infinite sequence of inputs without outputting. To make it true for that property we would need to use buffering that never allows an infinite sequence of inputs without outputs. This would be a strict refinement of the buffering constructions we are using and is only representable in models that handle unbounded nondeterminism.

Part (b) is straightforward and analogous to earlier results. ∎

The following lemma is a corollary to the previous two.

LEMMA 2.4 *If each process in a network N is accepting or has the finite output property, then so does the process $N^{\diamond\diamond}$ in which an arbitrary selection of the channels have had buffering added.*

In receptive processes, arbitrary buffering of an input channel is equivalent to putting an infinite buffer on that channel.

LEMMA 2.5 *If c is an input channel of the receptive process P then $\diamond c\!> P = \infty c\!> P$ for c any input channel of the strongly accepting process P.* ∎

PROOF It is obvious that

$$\diamond c\!> P \sqsubseteq \infty c\!> P = (B^{\infty}_{\langle\rangle}[out \leftrightarrow c]P)[\![{}^{c}/in]\!]$$

since the buffering on the left-hand side may choose never to refuse any input.

It is also easy to show that the traces of the two sides are equivalent. Where, on the left-hand side, the input buffering chooses to synchronise an input between the environment and P when its queue is empty, the right-hand side can simulate this via the (empty) buffer inputting and immediately outputting before any other global event has occurred.

As P never refuses inputs, $\diamond c\!> P$ could never be stable when its input buffer is nonempty. For it to be refusing inputs the buffer must be empty – and P must be refusing inputs with the buffer deciding on synchronisation. But P never refuses inputs, so we can conclude that $\diamond c\!> P$ is receptive, as indeed is $\infty c\!> P$.

It follows that the only way in which the two processes could differ would be for $\diamond c\!> P$ to have a refusal (s, X) with $X \subseteq O$ (necessarily with empty input buffer) that the right-hand process does not. This cannot happen since the difference in the input buffers neither has any direct effect on the output refusals; nor can it affect whether the combination is stable. ∎

We conclude this section with an important structural result about buffer tolerance. It is directly analogous to Lemma 2.2.

Theorem 2.6 *Suppose N is a network in which there are no cycles in the directed graph of channels.*

(a) If each individual process of N is leftward buffer tolerant, or each is rightward buffer tolerant. Then N has the same property (with respect to all channels).

(b) Suppose the subnetwork $M \subseteq N$ is convex, meaning that any path from one member of M to another using the directed channels lies entirely in M, and that every path from an input to an output through N contains a node in M. Then N is partitioned into those nodes before M, M itself, and those after M. Suppose also that each "before" node is leftward buffer tolerant, and that each "after" one is rightward buffer tolerant. Then N is weakly buffer tolerant with respect to all channels other than the internal channels of M.

proof For part (a) we prove the leftward result. The other is symmetric.

We can build N up a process at a time in such a way that if there is a channel from P to Q then Q is added before P. Claim that each of the resulting partial networks N_i are leftward buffer tolerant. It is plainly true for the first such network, which consists of only one leftward buffer tolerant process. So suppose it is true of N_i. Then, by induction, $N_{i+1}^{\diamond\diamond}$ is equivalent to the network $N_{i+1}^{\dagger}$ in which there is arbitrary buffering on all input channels of N_i and all channels of the added process P (noting that the channels from P to N_i fall into both these categories). The associative properties of parallel mean that this is equivalent to the parallel composition of *buff*(P) and N_i with buffering added just to *Inchans*(N_{i+1})$\cup$*Inchans*(N_i). The leftward buffer tolerance of P then gives us that this is equivalent to *inbuff*(N_{i+1}) as required.

Therefore *inbuff*$(N) = N^{\diamond\diamond}$, giving us the result we wanted.

Part (b) has a very similar proof, where nodes before M are added as above, and ones after it are added so that if there is a channel from P to Q then P gets in first. ∎

This is evidently a rather fundamental theorem where it can be applied. Most of the rest of this paper is devoted to establishing just when this is the case, and finding alternative arguments when it is not.

3 The buffer tolerance of chains and trees

As shown in [22, 18] and elsewhere, in deciding deadlock freedom it is crucial to know whether or not the target network is a tree, since it is a great deal easier to prove the property for networks with no cycles in their (undirected) communication graphs. In this section we will see that the same is true of buffer tolerance.

Chains

In Chapter 5 of [18], the author showed that any network of the form $P \gg Q$ (and therefore an arbitrary-length chain of processes) is buffer tolerant relative to the specification of being a buffer, under the assumption we are making of no selective input. That is the result encapsulated in "Buffer Law 6": if P and Q are processes with one output and one input channel each, and satisfy the no-selective-input assumption we make in the present paper, then

$$P \gg Q \text{ is a buffer} \Rightarrow P >\!\!\diamond\!\!> Q \text{ is a buffer}$$

In this section we will examine the extent to which this property can be extended to more general specifications. It would be nice if the following question had the answer "yes".

QUESTION 3.1 *Suppose $C = P_1 \gg P_2 \gg \ldots \gg P_n$ where each P_i has precisely the two channels $\{in, out\}$ and each P_i is deadlock-free and has the* *finite output property. Then is C weakly buffer tolerant in respect of all its channels?*

Unfortunately the general answer to this question is "no", as demonstrated by the following example.

EXAMPLE 3.1 Let P be the deterministic buffer which can reject an input only when both the following two conditions are true.

- It is non-empty, and
- it has output at least two items.

And let Q be the deterministic buffer which behaves like $COPY$ for one cycle and then becomes the infinite buffer $B^{\infty}_{\langle\rangle}$. Namely the only point at which it can refuse input is when it has one item in and has never output.

Consider $P \mathbin{>\!\!\diamond\!\!>} Q$. It can input three items (with no output), then refuse to input. For P might have output two of these items to the internal buffer, which in turn has output one to Q. All three processes are then capable of refusing input. $P \mathbin{>\!\!\diamond\!\!>} Q$ therefore has the failure $(\langle in.a, in.a, in.a\rangle, \{\mid in \mid\})$ for any member a of the type being transmitted.

Now consider $\diamond\!> P \gg Q \diamond\!>$. It can, in fact, never refuse any input since to do so P would have had to have output two things to Q, which means that Q does not refuse to input. Therefore in any stable state beyond this point P would have to be refusing to output, which implies that it is not refusing to input. Since $P \gg Q$ is the infinite deterministic buffer it follows that $\diamond\!> P \gg Q \diamond\!>$ is also, and so does not have the failure $(\langle in.a, in.a, in.a\rangle, \{\mid in \mid\})$.

Theorem 2.6 (b) means that we can conclude P is not leftward buffer tolerant, and that Q is not rightward buffer tolerant.

However $P \mathbin{>\!\!\diamond\!\!>} Q$ is indeed a buffer, as implied by Buffer Law 6 as quoted above. ∎

What we will do here is to concentrate on this type of chain with a single channel in the direction of increasing indices from P_i to P_{i+1} where the processes have the finite output property and are deadlock-free (or, equivalently, because of the single input channel and no-selective-input, accepting). The following results will essentially show that examples like the above where the buffered chain can refuse to input when $buff(C)$ cannot are the only ways in which weak buffer tolerance can fail.

Note that, by our earlier results, any chain made from deadlock-free and FOP processes itself has those properties, which gives us a reasonably strong liveness property in itself. The following result means that there is nothing to do in the traces model, though P and Q of the above example show that we cannot extend it to the stable failures or failures divergences model.

THEOREM 3.2 *(i) If P is any process with a single output and a single input channel, then P is leftward buffer tolerant in the traces model.*

(ii) If P is any deadlock-free, FOP process with a single input channel and a single output channel, then P is rightward buffer tolerant in the traces model.

(iii) Any chain of processes, each with a single input and a single output channel, is leftward buffer tolerant, and is rightward buffer tolerant if all the component processes are deadlock-free and FOP.

PROOF Note that (iii) follows directly from (i), (ii), Theorem 2.6 and Lemma 2.2, so all we have to do is to prove (i) and (ii).

For (i), what we have to prove is that $P\diamond\!> \sqsupseteq_T \diamond\!> P$. In other words $\mathit{traces}(P\diamond\!>) \subseteq \mathit{traces}(\diamond\!> P)$. To demonstrate this we show that $\diamond\!> P$ can simulate $P\diamond\!>$. Suppose t is any trace of $P\diamond\!>$: we can look at how this came about by knowing there is a trace s of $P\diamond\!>$ without hiding the outputs from P to the output buffer which explains t. Because we are only operating in the traces model we can assume that s contains no zero-buffering – all outputs from P go into the buffer strictly before being transmitted to the outside world.

What we will do is show how to model each action of s as $P\diamond\!>$ follows its *trajectory* (sequence of actions and operational states) by an action of $\diamond\!> P$. We do this in such a way that P_S (the simulating P) follows the same trajectory as P_O (the original P from $P\diamond\!>$) but and is always in the same state or behind P_O in the trajectory: specifically it is either in its initial state or in the state where it has just performed the most recent overall output which has appeared in t. The input buffer of $\diamond\!> P_S$ contains all the inputs P_O did between making the most recent output (if any, otherwise its initial state) which has so far appeared as an overall output in t, and the present. This description of how the simulation and original states tie together is the *coupling invariant*.

(a) An input action (by P_O) is mirrored by an input action by the buffer in $\diamond\!> P_S$.

(b) An output action by P_O to its buffer does not require any fresh action in the simulation.

(c) An output action by the buffer in $P_O\diamond\!>$ means that P_S has to advance to the state where it can perform this action, and then perform it. It does this by consuming (from its input buffer – hidden actions) all the inputs P_O did between its previous output (or initial state) and the current output.

Each of these clearly preserves the coupling invariant, and the existence of this *lazy* simulation proves (i).

For (ii), we have to prove the opposite containment. The condition that P is deadlock-free and has FOP is necessary, as is demonstrated by processes like $STOP$ and $\mu\, p.out \to p$. Neither of these can input, but obviously $\diamond\!> P$ can always choose to input.

Again suppose we are trying to simulate the trace t of $\diamond\!> P$ which has been expanded to s by un-hiding.

We set up an *eager* simulation of $\diamond\!> P$ by $P\diamond\!>$ whose coupling invariant is as follows: (i) the trajectory of P_S is always a (not necessarily strict) extension of that of P_O such that P_S has already input all those values input by the input buffer in the original; (ii) up to the point where that is finished, P_S's trajectory follows the trajectory P_O performed over the whole of s; (iii) the output buffer of the simulation contains all those values which P_O produced after the most recent output from $P\diamond\!>$ in the original and the present (in consequence of its additional inputs over and above what P_O has seen); (iv) $P_O = P_S$ (they have reached the same point in their trajectory) if and only both their buffers are empty.

This simulation works as follows. Once again we can – and do – assume that all inputs of the original system are actually held by the buffer at some point.

(a) P_S is initially in the initial state of P. Since this is the same as the initial state of P_O it satisfies part (iv) of the invariant.

(b) If P_O performs an output o, then either (thanks to P_S being ahead on the same trajectory) that output will already be present in the output buffer of $P_S\diamond\!>$, or P_S has only made the outputs that P_O has. In the first case the simulation outputs from its buffer. In the second case, the input buffer of $\diamond\!> P_O$ must be empty since otherwise P_S (having already performed these extra inputs) must have output o contradicting the emptiness of its buffer. We can conclude that, the input buffer of $\diamond\!> P_O$ is empty, and so $P_S = P_O$: we can therefore simulate o by P_S performing o (either through zero buffering or input and immediately output by P_S's output buffer).

(c) If P_O performs an input from the buffer, then because the buffer is non-empty there is no need for P_S to do anything to preserve the invariant, and so it does not.

(d) If the input buffer of P_O performs an input, then our assumptions about P mean that, either further down the trajectory of P_O or beyond it, P_S must come into a state where it can perform an input (an arbitrary input by no-selective-input). We simply move P_S to this state, absorbing all outputs that appear from P_S in its the output buffer.

Again, the existence of the simulation proves the required trace inclusion. Note that (i) and (ii) together prove that $\diamond\!> P$ is trace-equivalent to $P\diamond\!>$ for any deadlock-free FOP process P. This concludes the proof of Theorem 3.2. ∎

Turning to failures refinement, there are two approaches we can take: finding out what is provable for the general case (modulo conditions like acceptingness), and identifying conditions on individual processes which make them leftward or rightward buffer tolerant in that model, so we can use Theorem 2.6.

Assuming deadlock freedom (equivalent here to acceptingness, a property inherited by $\diamond\!> P$ and $P\diamond\!>$ thanks to Lemma 2.3) has one great advantage when considering processes with one input and one output channel. That is, it cannot refuse both channels at once: we can thus hope to be able to consider only failures of the form $(s, \{\mid out \mid\})$ and $(s, \{\mid in \mid\})$ and rely on no-selective-input and output decisiveness to fill in the rest. That would be very useful since the two sorts of refusal are going to be generated at opposite ends of a pipeline and we would not have to align them.

Despite this, the simulations which demonstrate trace containment do not work in many cases of failures refinement. (We will ignore divergences since we will usually be assuming conditions such as FOP which eliminate it.) Two reasons for this are:

- The output buffer can capture any output which is available from P in $P\diamond\!>$ and offer that item stably and alone, even though P might be unable to offer that output stably (it may only be offered as an alternative to τ, or in conjunction with inputting).
- The input buffer can refuse input in $\diamond\!> P$ just when P refuses it, and since the P of $P\diamond\!>$ is in general at a different state as that in its lazy simulation $\diamond\!> P$, we cannot necessarily expect their input offers to co-incide.

EXAMPLE 3.2 The process

$$\begin{array}{rcl} E & = & inp \rightarrow inp \rightarrow \\ & & \quad (outp \rightarrow outp \rightarrow outp \rightarrow E \;\Box\; inp \rightarrow E) \\ & & \Box\; outp \rightarrow inp \rightarrow E \end{array}$$

is neither leftward nor rightward buffer tolerant with respect to the failures model: $(\langle inp, inp\rangle, \{inp\})$ is a failure of $E\diamond\!>$ but not of $\diamond\!> E$, and $(\langle inp, inp, outp, outp, inp\rangle, \{inp\})$ is one is one of $\diamond\!> E$ but not of $E\diamond\!>$. (In both cases the buffers hold one item at the point of the stable refusal. This example was discovered using FDR.) ∎

Nevertheless there are several useful results we can obtain about buffer tolerance in the failures model. There is no reason why we should necessarily expect to get symmetric results for leftward and rightward buffer tolerance in this model, because the failures behaviour of general buffering looks very different on its two sides. This seems to mean that there is little we can usefully say about rightwards buffer tolerance. So we will concentrate on leftward: we might ask for what sort of P do we get $\diamond\!> P \sqsubseteq P\diamond\!>$?

In order to get this behaviour we need the *strong* form of output decisiveness. For example the process $NOD = (out \rightarrow out \rightarrow COPY) \rhd COPY$ is not output decisive, and $NOD\diamond\!>$ has the failure $(\langle\rangle, \{in\})$ whereas $\diamond\!> NOD$ does not: in the first the unstable offer of out by NOD is converted into a stable one by the output buffer, which might refuse the second out.

As demonstrated in the following result, we need to strengthen this further to state that any output can be made unconditionally (i.e. not as an option with an input, though it does not ban joint offers or inputs and outputs as additional nondeterministic choices).

THEOREM 3.3 *Suppose that P is a deadlock-free FOP process with one input and one output channel, and has the property that, whenever $s\hat{}\langle o\rangle \in traces(P)$ for o an output, then $(s, \Sigma \setminus \{o\}) \in failures(P)$ (*unconditional output*). Then P is leftwards buffer tolerant in the failures and failures/divergences models.*

PROOF Both $\diamond\!> P$ and $P\diamond\!>$ are divergence free by Lemma 2.3, and we know P is leftward buffer tolerant in the traces model. Therefore we can concentrate solely on the refusal components of failures. Observe that $\diamond\!> P$ itself satisfies *unconditional output* since it can refuse to input whenever P does.

Suppose $(s, X) \in failures(P\diamond\!>)$ is a maximal failure. We must show that $(s, X) \in failures(\diamond\!> P)$. Let us first consider the case when $\{\mid out \mid\} \subseteq X$: in other words $P\diamond\!>$ is refusing to output. Deadlock freedom then implies that $X = \{\mid out \mid\}$. This can happen only when the output buffer is empty (because nonempty buffers cannot refuse to output). In the lazy simulation of $P_O\diamond\!>$ by $\diamond\!> P_S$, the emptiness of the former's output buffer means that P_S has already produced all the outputs it will by the time it reaches the later state P_O. We can therefore advance P_S by having it take in the entire contents of its input buffer, which will bring it into the same state as P_O, without generating any external actions. (This can be achieved because of the coupling invariant we established earlier.) We know that P is refusing to output in $P\diamond\!>$, so the same state is refusing $\{\mid out \mid\}$ in $\diamond\!> P$. The state of $\diamond\!> P$ is stable because P is stable and the input buffer is empty.

So we can assume that $\{\mid out \mid\} \not\subseteq X$ and that $o \in \{\mid out \mid\} \setminus X$ is an output that the simulation can perform after s. Then, in the lazy simulation, either the buffer of $P\diamond\!>$ is nonempty after s (with first element generating o) or it is empty and P is in a stable state ready to output. In each case we cannot be sure that the particular trajectory T that P_S is following it the one we require. *Unconditional output* tells us that there is a trajectory T_o where it follows exactly the same trace and then offers *only* o stably. So we run the lazy simulation with T in the original and T_o in the simulation, which is possible since the P_S (up to the trace $s\hat{}\langle o\rangle$) never gets past the end of T_o. Just before P_S communicates o in T_o we know that it is stable and refusing all input. It follows that $\diamond\!> P$ is stable and can refuse all input (by the properties of arbitrary buffering). Since it is refusing all but o it follows that $\diamond\!> P$ has the required failure (s, X). ∎

The author has not been able to find any useful analogue in the other direction: notice how the asymmetry of buffering (able to refuse input but not output) is used crucially in the above proof.

Notice also that the proof that the case where all outputs are refused is much easier than the other part. The only place *unconditional output* is actually required is to prove that input refusals are present where necessary in the simulation (though it might be necessary to run the simulation further to get a stable state without it). For the output component of refusals we could make do with the weaker condition *strong output decisiveness*. The following result shows two ways of obfuscating any differences in input refusals like those that appear in Example 3.1.

THEOREM 3.4 *Suppose P is a deadlock-free FOP process with one input and one output channel, which is strongly output decisive. Then, with respect to failures,*

(i) $Chaos_{\{|in|\}} \underset{\{|in|\}}{\|} (\diamond\!> P) \sqsubseteq Chaos_{\{|in|\}} \underset{\{|in|\}}{\|} (P\diamond\!>)$

(ii) $\infty\!> P = \infty\!> P\diamond\!>$

The second of these easily extends to any chain of processes by induction.

COROLLARY 3.5 *If $C = P_1 \gg \ldots \gg P_k$ is a chain processes satisfying the above assumptions, then $\infty\!> C = \infty\!> C^{\diamond\diamond}$.*

There is a corresponding generalisation of (i), but the fact that strong output decisiveness is not preserved by chaining means that it cannot be proved in the same direct way. However the need for strong output decisiveness (as opposed to the ordinary sort) can be eliminated provided that there is a further process in the chain.

THEOREM 3.6 *(a) Suppose P is a deadlock-free FOP process with one input and one output channel, and that Q is another such process. Then $Chaos_{\{|in|\}} \underset{\{|in|\}}{\|} (\diamond\!> P \gg Q) \sqsubseteq Chaos_{\{|in|\}} \underset{\{|in|\}}{\|} (P >\!\!\diamond\!\!> Q)$*

(b) *If $C = P_1 \gg \ldots \gg P_k$ is a chain of deadlock-free FOP processes, then* $Chaos_{\{|in|\}} \underset{\{|in|\}}{\|} (\diamond\!> P) \gg Q \sqsubseteq Chaos_{\{|in|\}} \underset{\{|in|\}}{\|} (P\diamond\!>) \gg Q$

(c) *If $C = P_1 \gg \ldots \gg P_k$ is a chain of deadlock-free FOP processes satisfying, then $\infty\!> C = \infty\!> C^{\diamond}$.*

(b) follows inductively by (a), and the proof of (a) is via a lazy simulation very like the ones already used. Since all output refusals are now produced by Q, we do not need strong output decisiveness of the process (P) we are pushing the buffer through. (c) is a variant of Corollary 3.5 where we do not have to worry about output refusals appearing from buffers.

Trees

It seems natural to move from chains to trees: networks with no cycles in their undirected communication graphs. To see that an undirected cycle is important, suppose that there are two disjoint paths for data to reach P_j from P_i. It is obvious that adding buffering can alter the order in which information arrives over the two routes. For example, the original network might ensure that the number on route A never exceeds the number on route B. P_j may break if this invariant is violated, which it might easily be if buffering were introduced on route B. Specifically, P_j may reach states with the buffering that it can never reach without. In other words it may fail local buffer tolerance.

Note that Theorem 2.6 covers networks with this sort of structure, but the requirements of that result mean that behaviour like that described above would be banned. If P_j were leftward buffer tolerant it would not maintain $\#A \geq \#B$, and if P_i were rightward buffer tolerant then failure of this property of its inputs could not lead it to any new states.

Example 3.3 Trees with more than one output channel will require output buffers: consider the network T in Figure 1.

Suppose P is a process that, takes in inputs and outputs each once along each of the internal channels:

$$P = in?x \to a!x \to b!x \to P$$

The other two processes are both $COPY$. If we place buffering on the internal channels then there is no relationship between the number of outputs from

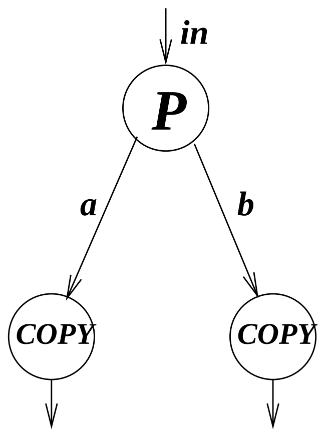

Figure 1: Illustrating the need for output buffers

the two $COPY$s. If, on the other hand, we simply place a buffer on P's input channel (analogously with our chain results), the numbers of outputs on the two channels never differ by more than 2: so it is not leftwards buffer tolerant. If we place buffers on all three external channels then the internally buffered version's behaviours are all present in $buff(T)$. ∎

Example 3.4 Figure 2 illustrates two versions of a simple two process tree: one with internal buffering and one with external. (Asterisks denote buffered channels.) Process P has one input channel and two output channels, while F has two inputs and one output (though the example would also work when F has no output channels). P is the same as P from Example 3.3, while

$$F = d?x \rightarrow a?y \rightarrow e!x \rightarrow F$$

So the network has input channels in and d, and output channels b and e.

In the internally buffered case it is possible for outputs on b to occur any number of times without any inputs on d, since the outputs P makes along a can be buffered. This is impossible in the externally buffered case since the number of outputs on b never greater than the number of communications on a, which is in turn never greater than the number of communications along d.

Clearly F is not accepting, and this fact is crucial in making this specific example work, but hopes that we might eliminate this problem by making

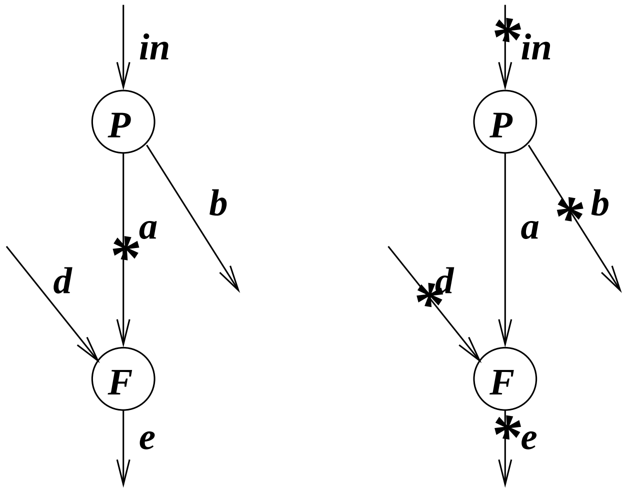

Figure 2: A tree network which is not buffer tolerant

processes accepting are dashed if F is replaced by $F \mid\mid\mid RUN_{\{|d,a|\}}$. This is clearly a accepting process but in any behaviour where the number of outputs equals the inputs on its two channels, all communication must have been performed by the parallel component F. It follows that in any behaviour of the externally buffered network which ultimately outputs as many things on e as it has input on in, inputs from d must have occurred before outputs on b.

Therefore this two process network is not even weakly buffer tolerant in the traces model. ∎

We therefore cannot hope to get a result which proves as much about general trees as we have established about chains. It turns out that we can still, however, do quite well.

What turns out to be crucial in Example 3.4 is that there are two disjoint routes from its inputs to its outputs (namely in, b involving only P and only F through d, e).

We can get close to our tree results if we avoid this possibility: the most obvious way is to use only *fan-in* nodes (ones that have precisely one output channel) or *fan-out* ones with precisely one input channel. We can generalise respectively the lazy and eager simulations to these categories. We first deal with the fan-in case.

THEOREM 3.7 *(i) If P is any fan-in process, then P is leftward buffer tolerant in the traces model.*

(ii) If P is any deadlock-free and FOP fan-out process then it is rightward buffer tolerant in the traces model.

PROOF For part (i), the lazy simulation carries across to this case, the only amendment being that external inputs are now made into (perhaps) more than one buffer, and P_S may have to input from more than one buffer before it can simulate the next output.

Similarly, the eager simulation carries across for part (ii). (Recall that under our standard assumptions a process with one input channel is deadlock free if and only if it is accepting.) ∎

The reasoning behind Theorems 3.3 and 3.4 also carries across to this case, the proof being essentially the same.

THEOREM 3.8 *Suppose that P is a deadlock-free FOP fan-in process.*

(a) If P has the unconditional output *property described earlier then it is leftwards buffer tolerant in the failures and failures/divergences model.*

(b) If P is strongly output decisive and its input events are I, then

(i) $Chaos_I \underset{I}{\|} (inbuff(P)) \sqsubseteq Chaos_I \underset{I}{\|} (P\diamond\!\!>)$

(ii) $inbuff^{\infty}(P) = \infty\!\!> P\diamond\!\!>$

The analogue of Theorem 3.6 also holds, where now there may be many P's: one for each input channel of Q.

We cannot expect leftwards buffer tolerance to hold for fan-out processes, since a range of buffers on the output channels have much more freedom to vary the order of such a P's communications than a single input one. For example the latter could not interfere with the order in which P outputs along its various channels (say alternating between two), whereas output buffers certainly could.

For similar reasons rightward buffer tolerance does not hold for fan-in processes. We can, however, get the following result.

THEOREM 3.9 *Suppose we have a tree network T where (i) every node has at least one input and at least one output, is deadlock-free and FOP, and (ii) there a node X which is on every path from an input channel T to an output channel of T. Then T is weakly buffer tolerant in the traces model.*

PROOF Every node reachable from X (downstream) can only have one input channel (or else there would be a cycle in the undirected graph involving this node and X). So it is a fan-out process. And every node from which X is reachable (upstream) is fan-in for similar reasons. We can therefore apply Theorem 2.6 (b) with $M = \{X\}$. ∎

Unfortunately, Example 3.4 shows that we cannot hope to get directly analogous results for a *general* network where there is both fan-in and fan-out branching. In particular we may have to make extra inputs to $\mathit{buff}(T)$ to get the outputs which appear in $T^\diamond$. The following result shows that this is sufficient and that the values of the extra inputs are irrelevant. What it says is that, provided all the nodes have a property which means that they can always eventually make progress on each channel, and there are no cycles in the communication graph, then every pattern of outputs which can appear from $T^\diamond$ can appear in $\mathit{buff}(T)$ if sufficient extra inputs are entered on each channel.

THEOREM 3.10 *Suppose T is a network whose undirected communication graph is a tree and whose nodes P all have the following properties:*

- *Deadlock-free.*
- *Fairness: Every infinite trajectory of P contains infinitely many states which can communicate on any chosen channel (input or output).* Note that this is weaker than saying that each infinite trace contains infinitely many communications on each channel.

Suppose also that all internal communications of T are hidden. Let t be a trace of $T^\diamond$. Then there is a trace s of $\mathit{buff}(T)$ such that

(i) For each input channel c we have $t \downarrow c \leq s \downarrow c$. Let $s \downarrow c = (t \downarrow c)\hat{\ }e_c$.

(ii) If O is the set of all output events, then $s \restriction O = t \restriction O$.

(iii) The truth of the above does not depend on the precise value input during the extra inputs e_c for any c, though the required number of subsequent additional inputs required may change.

PROOF The proof of this result is via another 'eager' simulation. In this we will make sure that every node is always at least as far advanced along its trajectory (or beyond it) as is in the original trace t of $T^\diamond$. We can again use the output buffers of $\mathit{outbuff}(T)$ to hold all outputs until the correct time to release them (or not at all if they were output after the outputting node went beyond the end of its trajectory in t). The only difference with the simulation used in the proof of Theorem 3.7 is that this time, in order for one node P making an output to advance, it may be be necessary for the node Q which absorbs that output to move into a state where this can happen.[5] This might well mean that Q needs further inputs. The following is a sketch of the proof.

The result (and in effect the simulation) is built up inductively over the structure of T, adding on a node at a time in such a way that the new node has no input channels from the ones that are already there. What we prove for each of these subnetworks S is that, for arbitrary $n_c \in \mathbb{N}$ for each $c \in \mathit{Inchans}(S)$:

- $\mathit{outbuff}(P)$ can simulate $S^\diamond$'s behaviour along T in such a way that each of its nodes is always at least as advanced as the corresponding node in $S^\diamond$, and such that *at least* n_c additional inputs of whatever values we please have been accepted on each $c \in \mathit{Inchans}(S)$.

- The outputs in the simulation follow the same pattern as in t, but some inputs may have been advanced to before the corresponding point in t.

This is trivial for a single node: once the end of its t-trajectory is reached the condition above implies that it will accept any number of inputs we wish on each channel, and since we may hold any post-trajectory outputs in the output buffers of $\mathit{outbuff}(P)$ there is no need for them to appear externally.

So suppose it holds for the S to which we wish to add a new node P to get S^+. Note that

- The input channels of the combination are those of P plus all those of S that are not connected to P.

[5]This does not arise in the lazy simulations of Theorems 3.2 and 3.8, because there the simulations are driven by demand for outputs, and each P only has one. It happens only trivially in Theorem 3.7, since the conditions imply that we can always advance the recipient node without it accepting inputs from anywhere else.

- The output channels of the combination are all those of S plus all those of P that are not connected to S.
- The components S_c of S attached to different output channels c of P are otherwise all distinct (and disjoint).

Suppose we have requirements n_c for each of the input channels of S^+. For each of the channels c that connects P to S, inductively construct the simulation for the component S_c of S that is connected to c, using $n_c = 0$ and all other n_d being inherited from the main problem. Let m_c be the actual number of inputs consumed on c in this simulation, maximised over all choices of input values.[6]

Then we can (by fairness) run P beyond the end of its trajectory in t so that it exceeds n_b for each of its input channels b and has produced at least m_c more outputs than in t on each of its output channels c that is connected to S.

If P has produced more than outputs on some c than are required, then the simulation in S_c can be re-worked so that they are all consumed. It is vital here that the S_c are all different so that no contradictory requirements are generated.

The outputs produced by this can be absorbed by the output buffers and produced when required. ∎

4 Model checking properties of buffer tolerant systems

When it comes to the crunch, the real objective of this paper is finding practical ways to verify systems containing arbitrary buffering. In this section we provide some results and techniques that do this.

The following are corollaries to the simulation produced for Theorem 3.10, observing that the external buffers of *buff*(T) do not change what states can be reached by T and its nodes. They show that for an important but restricted class of specifications, the fairness principle and the lack of cyclic dependencies amongst the channels are sufficient to give strong buffer tolerance with respect all a network's channels.

[6]That such a maximum exists follows from König's Lemma.

THEOREM 4.1 *Suppose the network T satisfies the conditions of Theorem 3.10. Then, if P is one of the nodes of T which can reach state Q in $T^\diamond$, it can also reach Q in T (i.e. the network without any buffering). In other words, N is locally buffer tolerant.* ∎

THEOREM 4.2 *Suppose the network T satisfies the conditions of Theorem 3.10. Then if in some run of $T^\diamond$ the nodes P_i can attain (respectively) the states Q_i in some order, then the same is true of T (though not necessarily in the* same *order).* ∎

In FDR, reachability conditions are usually couched in terms of indicator events flagging a particular state. Since the above results apply to the states of individual processes in the network, they correspond to indicator events that are not synchronised between the processes. These results are so direct and simple that there is a substantial incentive to couch specifications in this form if possible.

Frequently, however, this cannot be done. However results in the previous section are nearly as useful. They imply a number of variations on weak buffer tolerance of different types of tree network (though not for the input refusals). Now observe that Lemma 2.1 allows us to conclude that, if $S \sqsubseteq N$ and is weakly, leftward, or rightward buffer tolerant in one of the standard models, then $N^\diamond$ refines a version of the specification S with appropriate external buffering. *This means that a finite-state check has verified that a potentially infinite state process meets a potentially infinite-state specification.*

This leaves us with three difficulties:

- Theorem 3.10 falls short of weak buffer tolerance, even in the traces model, meaning that the above observation does not apply to networks where nodes have both fan-in and fan-out branching. We can address trace specifications which are relevant to just one input or to just one output by cutting down the network to that part which feeds, or is fed by, the relevant channel (namely, cutting it down to a network meeting the conditions of Theorem 3.8 or 3.7. To complete this we abstract (by hiding) all the channels of the cut down network that are (respectively) outputs or inputs whose other ends have been removed.

- None of the results allow us to say much about the input refusals of our network. However we do have Lemma 2.2 which contains useful information about these, and is frequently sufficient, as does the result (see [18] for example), that any tree of processes where each adjacent pair are connected by a single no-selective-input channel is deadlock free if its component nodes are.

- It is perhaps not always obvious what the buffered specification means intuitively. We will go some way to understanding this problem in the first sequel to this paper.

Applications

The above discussion tells us that if P_i are deadlock-free FOP processes and $C = P_1 \gg \ldots \gg P_n$ then

$$S \sqsubseteq_T C \Rightarrow \diamond\!\!> S \sqsubseteq_T C^{\diamond}$$

This extends Buffer Law 6 from [18], quoted earlier, to any trace specification such that $\diamond\!\!> S = S$. Some examples of such specifications are:

- P is a bag (multi-set): it outputs each input once, but is allowed to re-order.
- Every output value of P was also an input value.
- Over the natural numbers, the output is always less than or equal to the sum of the inputs.
- The value *error* is never output.

Examples of systems to which we might apply the chain result include:

- Long-distance network communications where data is passed over a chain of nodes, possibly being modified or blocked. It is of course natural for buffering to be used in such cases.
- A pipeline operating on a stream of data, each passing its outputs on to the next. For example we might be applying various operations such as filtering and clipping to an audio or visual signal.

- Layered protocols, which can be represented (as discussed in [18]) by networks of the form

$$T_1 \gg \ldots T_k \gg M \gg R_k \gg \ldots \gg R_1$$

  in which the pairs (T_i, R_i) are complementary. Even when the overall effect is not a buffer (the case discussed in [18]) the entire system is now known to be buffer tolerant.

A *stack* process operates on the Last-In-First-Out (LIFO) principle as opposed to a buffer's FIFO. Imagine a series of buffers and stacks connected together in a pipeline: trying to think about what effects this might have on an input stream is somewhat baffling! The fact that it trace-refines just the stacks with arbitrary buffering added at the front (a consequence of Theorem 3.2) would certainly help given specific stacks.

The fan-in result can be used to reason about tree networks where a variety of inputs contribute to a single output. For example, it could apply to a network of merging or multiplexing processes, or to a pipeline which can take inputs at more than one point.

An example of the latter is a pipelined system which has an interrupt signal that is accepted by one or more of its processes, which causes it to go back to some state specified by the signal. Our analysis shows that the system with internal buffering refines the one where there are buffers on the main input channel and all the interrupt signals: see Figure 3.

The fan-out result works on the opposite sort of network, for example consisting of demultiplexers, and Theorem 3.9 can be used to handle networks where information is brought together to a single point and then distributed again, for example multiplexers followed some chain and then demultiplexers. Note that it can be combined with Lemma 2.2 to give liveness results for $T^{\diamond}$.

For an example of Theorem 3.10, consider the network in Figure 4. Here, there is a controller node C which inputs data and, on the basis of it, decides what is to happen to all the items passing down a number of data streams. Assume that a control signal appears on each output channel of C after every input it makes, and that the processing nodes N_i satisfy the fairness condition (and all are deadlock free). We can use Theorem 4.2 to show, for example that illegal combinations of states cannot be reached in the processes M_i.

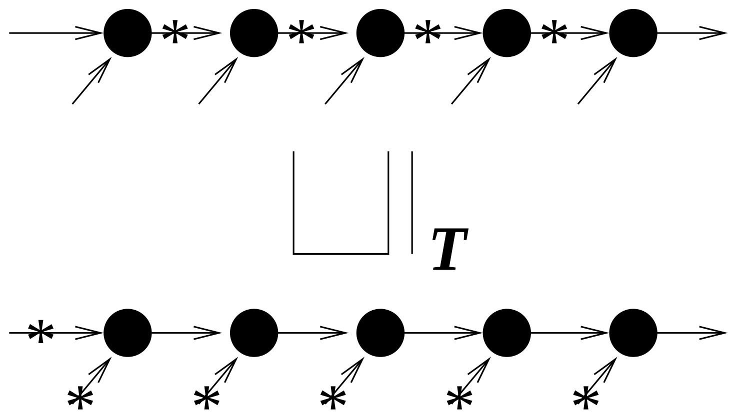

Figure 3: Interruptable pipeline.

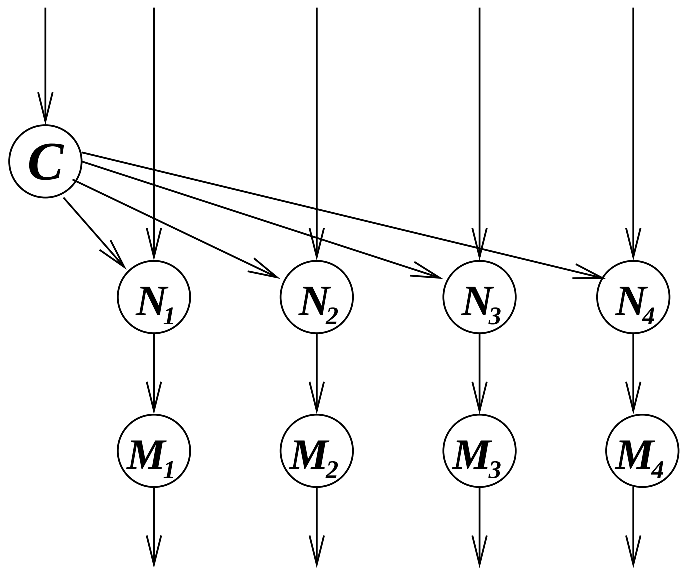

Figure 4: Process controlling multiple pipelines

Part 2: Functional and Confluent Behaviour

5 Specifying weakly buffer tolerant processes

Remembering that leftward and rightward buffer tolerance both imply weak buffer tolerance, many of the results in Part 1 were directed towards proving that $N^\diamond$ or similar refines *buff*(*Spec*) by proving that N refines *Spec*.

This begs the question of what *buff*(*Spec*) means for our specification *Spec*: when is it a useful thing to prove that $N^\diamond$ refines it? Another way to look at this is to ask for what specifications S can we find *Spec* (hopefully finite state) such that $S = \mathit{buff}(\mathit{Spec})$, since it is only these specifications that we can hope to prove that a weakly buffer tolerant system (containing buffering) meets. We first identify some things that S cannot do:

- It cannot specify that an input event on channel in_1 occurs before another one on channel in_2, since *buff*(*Spec*) can perform these in either order.

- It cannot specify that an output event on channel out_1 occurs before another one on channel out_2 since, whenever these two outputs can emerge from *Spec*, the buffers of *buff*(*Spec*) can produce them in either order.

- If $s\hat{\ }\langle o, i\rangle\hat{\ }t \in \mathit{traces}(S)$ for o an output and i an input then $s\hat{\ }\langle i, o\rangle\hat{\ }t \in \mathit{traces}(S)$ since the relevant input buffer can accept i just before the output if it can accept it just after.

- It can never limit what inputs can happen since these can always be accepted by the input buffers.

It follows that the traces of *buff(Spec)* are closed under reordering of (i) inputs on distinct channels, (ii) outputs on distinct channels, (iii) moving an input to before an output and (iv) adding arbitrary inputs at the end of a trace. These are exactly the conditions identified by Josephs in [8] as the ones which apply to the traces of a data-flow process (ones in which an infinite unbounded buffer is assumed on every channel). This is not in the least surprising since we may, of course, put such an infinite buffer onto the channels of $N^\diamond$. However our failures specification will in no way imply that infinite buffering is actually present.

This leads us to two observations which relate Joseph's work to our own and which are true at the level of trace equivalence:

(A) The trace specifications which make sense for weakly buffer tolerant networks are precisely the members of the semantic model for data-flow processes; one might therefore contemplate writing them in Joseph's version of CSP which makes explicit allowance for the closure conditions (i–iv).

(B) The members of the data-flow semantic model for traces are precisely $\mathit{buff}(P)$ as P varies over the ordinary traces model. (This was established by Josephs, Hoare and He in [9].)

We also observe that, for networks whose design is buffer tolerant in our sense, our work gives a model (namely the network with no buffering) for data-flow processes that can probably be used to model-check them, since it is likely to be finite state.

The above properties suggest that we might examine predicates which apply to the collection of input and output streams that a process has (i.e. the sequences of values that have passed along its channels). So suppose we have a trace predicate that can be written $R(is_1, \ldots, is_m, os_1, \ldots os_n)$ (is_i and os_j respectively being input and output streams) and which is closed under extension in input streams and prefix in output streams: $is'_1 \geq is \wedge \ldots \wedge is'_m \geq is_m \wedge os'_1 \leq os_1 \wedge \ldots \wedge os'_n \leq os_n$ and $R(is_1, \ldots, is_m, os_1, \ldots os_n)$ implies $R(is'_1, \ldots, is'_m, os'_1, \ldots os'_n)$. Suppose also that $R(\langle\rangle, \ldots, \langle\rangle)$. Then $\tilde{R}$, the largest prefix-closed set of traces which all satisfy it, trivially satisfies the conditions to be a (trace) data-flow process.[7] It follows that any trace property which can be written in the form $\tilde{R}$ for such an R is one we can hope to prove of a weakly buffer tolerant process since $\mathit{buff}(\tilde{R}) =_T \tilde{R}$. Let's call these properties *stream predicates*.

We should note that many stream predicates R will give rise to infinite-state CSP processes, which means it is not attractive to use them as specifications for FDR. What we know from Part 1, however, is that we can prove them for any weakly buffer tolerant network including buffering by proving that the network without buffering refines any (not necessarily data-flow) CSP process S such that $\tilde{R} \sqsubseteq_T S$.

[7] It is tempting to think that the reverse is true, namely that every data-flow process can be written as such a predicate. To see this is not true, consider the following processes

Note that the closure of stream predicates under extensions of input traces and prefixes of output traces mean that all they can constrain is what outputs are justified by some prefix of the current inputs. They must allow any input at any time and performing an extra input cannot ban a previously legal output.

6 Functional agents

Though much of what follows can probably be modified to encompass more general forms of stream predicate, it is interesting to concentrate on the subcase in which all of the output streams are in effect functions of the input streams. These cover a wide variety of applications and are conceptually simpler.

The relations they are based on will not be functions in the usual sense in which relations are considered to be functions, namely with each set of inputs corresponding to one sequence for each output channel. For we know they

---

P and Q (when there is only one input event i and one output event o):

$$\begin{aligned}
P &= (o \to i \to i \to P') \,\Box\, (i \to P_1)\\
P_1 &= (o \to i \to P') \,\Box\, (i \to P_2)\\
P_2 &= (o \to P') \,\Box\, i \to P_2)\\
P' &= i \to P' \,\Box\, o \to \mathit{IN}\\
\mathit{IN} &= i \to \mathit{IN}
\end{aligned}$$

$$\begin{aligned}
Q &= i \to Q_1\\
Q_1 &= (o \to Q') \,\Box\, (i \to Q_1)\\
Q' &= (o \to \mathit{IN}) \,\Box\, (i \to Q')
\end{aligned}$$

Each of P and Q satisfies all the conditions for being a data-flow process: applying arbitrary buffering to either side of either of them makes no difference. The standard distributivity properties of CSP operators imply that the same is true of $P \sqcap Q$.

Notice that in P the first output is available immediately, but the second one is not available until after two inputs; whereas in Q both outputs become available after a single input. Each of P and Q can be represented by a stream predicate, but $P \sqcap Q$ (also a data-flow process, since these are easily seen to be closed under nondeterministic choice) cannot. Clearly the predicate R would have to pass the traces $\langle o\rangle$, $\langle o, i\rangle$ and $\langle i, o, o\rangle$; but the streams of the last are the same as those of $\langle o, i, o\rangle$. Since these streams are what determines R, and since all the prefixes of $\langle o, i, o\rangle$ also satisfy R, it follows that this trace belongs to $\tilde{R}$. But this trace does not belong to $P \sqcap Q$.

have to be prefix-closed in outputs and extension-closed in inputs. Rather, we will specify that for each set of inputs there is a single set of notional outputs of which the actual outputs are *prefixes*.

If $is_1, \ldots, is_m$ are the input streams, there will thus be a function $f_d(i_1, \ldots, i_k)$ for each output channel $d \in \mathit{Outchans}$ (corresponding to os_r, say) such that

$$R(is_1, \ldots, os_n) \equiv \forall\, d \in \mathit{Outchans}.os_k \leq f_d(is_1, \ldots, is_m)$$

(Here, $s \leq t$ is the prefix order on traces: s is an initial subsequence of t.) For this to make sense we require that all of the functions are monotonic, namely whenever $is_r \leq is'_r$ for all r then $f_d(is_1, \ldots, is_m) \leq f_d(is'_1, \ldots, is'_m)$ for all d.

This concept makes sense if we were to allow $f_d(is_1, \ldots, is_k)$ sometimes to be an infinite sequence. This corresponds to a system that can generate an infinite number of outputs for a finite set of inputs. Of course if this value is infinite then extending the is_j will not, thanks to monotonicity, change the output available on d.

A stream predicate will be said to be *functional* if it is based on such a relation, and *weakly functional* if it is the subset of such a relation. For any weakly functional predicate there is a least system of functions (i.e. one whose results are always prefixes of all the others) which determines it.

As far as traces are concerned, we will say that a process is a *functional agent* (or just *functional*) if its traces all satisfy some functional stream predicate. (Note that since traces cannot express liveness properties, it does not make sense to differentiate between the weak case and the full case of functionality.)

Normally, of course, such agents will have at least one each of input and output channels, but the definitions we state do make sense when there are none of one or the other.

There is a natural way to turn this property of processes into a failures specification which is related to the standard specification of a buffer:

(A) If, for the given trace t, at least one of the output streams is incomplete, namely $t \downarrow d < f_d(is_1, \ldots, is_m)$, then $(t, O) \notin \mathit{failures}(P)$. In other words, if according to the functions some output is pending then the process cannot refuse to output.[8]

[8] On balance this definition seems preferable to insisting that the particular output

(B) If, for the given trace t, $t \downarrow d = f_d(is_1, \ldots, is_m)$ for all d, then $(t, I) \notin$ *failures*(P). In other words inputs on at least one channel are accepted when no outputs are possible.

We will, in fact, tend to be more specific than (B) here about what inputs are accepted.

The following are some examples illustrating functional agents.

(i) Notice that a process with one input and one output channel is a buffer (in the usual CSP sense) if and only if it is functional using the identity function.

(ii) A process with one input and one output channel which duplicates each input is described by the function $f(\langle\rangle) = \langle\rangle$, $f(\langle a \rangle \hat{}\, s) = \langle a, a \rangle \hat{}\, f(s)$.

(iii) A process which alternately merges two input channels into one is described by

$$f(\langle\rangle, t) = \langle\rangle \quad f(\langle a \rangle \hat{}\, s, t) = \langle a \rangle \hat{}\, f(t, s)$$

(iv) A process which splits its input stream into two is described by the pair of functions *odds* and *evens* defined:

$$\begin{aligned} odds(\langle\rangle) &= evens(\langle\rangle) = \langle\rangle \\ odds(\langle a \rangle \hat{}\, s) &= \langle a \rangle \hat{}\, evens(s) \\ evens(\langle a \rangle \hat{}\, s) &= odds(s) \end{aligned}$$

(v) A binary comparator, which inputs values on two input channels, compares then, and outputs the greater and lesser respectively on its two output channels, is a functional agent, as are elements like multiplexers and de-multiplexers.

An obvious hope is that any network composed of functional agents is functional. We must break this up into two parts: networks with directed cycles and ones without. It is obvious that any network of the latter (acyclic) sort is trace functional if all its components are: the function simply being

channel cannot be refused: we allow the process to decide what order to produce its pending outputs in.

the composition implied by the way the network together of the functions underlying its components.

It is also fairly obvious that any general network of functional agents (one which might have cycles) always has traces on each internal and output channel which are prefixes of the least fixed point of the monotone function from tuples of traces to tuples of traces which the structure of the network determines, for the particular set of input channel values we are given. In other words, if $is_1, \ldots, is_n$ are the sequences of values available on the network's input channels (all necessarily input channels of individual processes within it), and $cs_1, \ldots, cs_m$ are the rest of the channels, we know that each node, P_i determined by functions $f_{i,1}, \ldots, f_{i,r}$ for its r outputs, forces

$$cs_{s_{i,j}} \leq f_{i,j}(ds_{i,1}, \ldots, ds_{i,s_i})$$

for each cs_{s_j} which is an output of P_i, where the $d_{i,j}$ are those is_a and cs_b which are the inputs to P_i. If the inequalities above are turned into equations we get one equation for each internal channel: that channel is a monotonic function applied to a number of other internal and input channels. Standard fixed point theory then says that this system of equations has a least solution in the space of assignments of the internal channels to finite and infinite sequences, ordered by prefix. It is an easy induction on the construction of the fixed point and the evolution of the network that at all times the value of each internal channel is a prefix of this least fixed point. This is the same construction that is used in Kahn-MacQueen networks [10, 11], which are of course very similar. (The adherence of networks of this type to the least fixed point is sometimes called the *Kahn principle.*) One way of looking at a functional agent is as a functional node in such a network which is not forced to accept all possible inputs.

For any network of functional agents, we define its *principal* functions to be the functions, one for each output channel, of the input streams, generated by the composition or least fixed point described above.

Without any further restriction on the way the component processes behave even the acyclic case may well fail the failures specification of functionality. Consider, for example, the following processes which implement examples (iii) and (iv) above:

$$\begin{aligned} \mathit{Merge} &= a?x \rightarrow \mathit{out}!x \rightarrow b?x \rightarrow \mathit{out}!x \rightarrow \mathit{Merge} \\ \mathit{Split} &= \mathit{in}?x \rightarrow \mathit{odd}!x \rightarrow \mathit{in}?x \rightarrow \mathit{even}!x \rightarrow \mathit{Split} \end{aligned}$$

If these are connected by synchronising a with *odd* and b with *even* then they implement the expected identity function. If, on the other hand, we synchronise a with *even* and b with *odd* then the combination deadlocks rather than computing the principal function which is a delay-by-one of the identity function.

The conclusion is that in order to get functional behaviour in the acyclic case we must do enough to prevent the network from deadlocking, though that is not always sufficient since complex patterns of dependency can arise. This topic is closely related to the study of deadlock more generally. We leave a detailed study to a later date, but conclude this section with the following results, most of which are related to the author's earlier work on deadlock.

Recall that a *accepting* process is one which in any stable state can either output or accept input on every input channel. This is by no means a universal property, but when true it is very useful. For example, while a non-accepting process may be functional with respect to two different sets of functions (which differ on input combinations that the process does not accept), a accepting one uniquely determines its functions provided it has the finite output property (meaning that it never gives an infinite sequence of outputs for a finite input).

THEOREM 6.1 *Suppose N is a network of functional agents which are all accepting and satisfy the finite output property. Suppose also that it is either acyclic or has the following properties.*

- *It has no* cycles of ungranted output requests. *In other words it cannot get into a state where each of a cycle of processes is each waiting to output to the next and has no external output. The concept of cycles of ungranted requests, together with many techniques for avoiding them, are discussed in [22, 18], for example.*

Then N is accepting and is functional with respect to its principal functions.

PROOF Since the acyclic case trivially implies that there is no cycle of ungranted output requests, we will consider only the general network case.

We establish a lemma, namely that, in any state where the overall network refuses all outputs, all constituent processes are (a) accepting all inputs and (b) refusing to output. Note that the assumption and (a) together imply (b) since if a node accepts an output then either the overall network does or there is an internal (τ) action available since the internal recipient accepts the output. We need therefore prove only (a).

If (a) does not happen then the node refusing some input must (as it is accepting) offer an output on some channel c_0. Necessarily c_0 is internal since the network offers no output. It follows that the internal recipient must be refusing c_0, and hence offering output on some channel c_1. Evidently we can continue this selection inductively, and it eventually repeats to create a cycle of ungranted output requests, which we have assumed to be impossible. So the lemma is proved.

It immediately follows that the network is accepting. It also follows that in any stable state *either* the network is prepared to output *or* for every single node in the network each of its output streams actually equals the corresponding function of its input streams. (If one of them were not equal, the failures specification of a functional agent says the node can output.)

In the case of an acyclic network, this trivially shows that the output streams of the network equal the composed functions of the input streams whenever output is being refused. In the case of the general network, it shows that the values of the streams represent a fixed point of the system of equations the network represents. As argued above this is necessarily the least fixed point.

In either case the network cannot refuse to output until the complete set of functional values we have predicted is output. This completes the proof.

This result begs the question of how to avoid cycles of ungranted output requests. This problem is very similar to that of avoiding cycles of ungranted requests in deadlock freedom (see [22, 18], for example). Many of the techniques used there, such as variant functions, should apply to this more restricted case. One obvious rule which guarantees their absence is provided by the following result. Here, a pair of *concyclic* channels are an input and an output channel of the same node that appear in the same cycle in our network.

PROPOSITION 6.2 *Suppose N is a network of accepting functional agents. Suppose furthermore that whenever i and o are respectively input and output concyclic channels of a single node, then that node (operating independently of the network) has never output more items on o than it has input on i. Then N is free of cycles of output requests.*

PROOF If such a cycle existed then we could immediately say that the streams on all the channels involved in the cyclic have the same length, as these lengths are non-increasing round the cycle. But then each node is (in itself) able to make a further output, contradicting the assumption.

It is interesting to ask what would happen in non-accepting networks. We have already seen (in *Split* and *Merge* in their reverse connection network) that deadlock can arise when a pair of channels join two processes in an acyclic network, but it is also possible when there is just one channel per pair. Imagine three left-hand nodes L_0, L_1, L_2, each with one input and two outputs, and three right-hand nodes R_0, R_1, R_2, each with two inputs and one output. There are channels from L_i to R_i and $R_{i\oplus 1}$. On receiving a value from its input channel L_i outputs it to R_i and $R_{i\oplus 1}$ in that order and goes back to its initial state. R_i expects values from $L_{i\ominus 1}$ and L_i in that order and outputs the one from L_i before returning to its initial state. Functionally, this network should map a triple of input streams to the same triple truncated to the length of the shortest. In fact it accepts one input on each and then deadlocks. A cycle of ungranted requests forms around the L_i and R_i. (This example was adapted from one in Chapter 13 of [18].)

Even in cases where deadlock is impossible, the network may not be functional and, if it is, it may be with respect to functions that sometimes deliver proper prefixes of the ones predicted by composition or least fixed points.

7 Are networks of functional agents buffer tolerant?

If P is a functional agent with respect to a particular set of functions, it is reasonably obvious that $P \diamond c>$ is also for any output channel c of P (with the same set of functions). This is because the combination can only refuse to output when the buffer is empty and P refuses to output. In the case of a non-accepting P, the same may not be true (in the failures model) for an input channel since the buffer may accept inputs which P could refuse for some time, during which time the outputs owed from these extra inputs could be refused.

Since we can build the internally buffered network $N^\diamond$ by adding buffering onto all the internal output channels, it follows that $N^\diamond$ is also a network of functional agents following the same functions as in N.

It immediately follows that N is strongly buffer tolerant with respect to the *trace* specification of being a functional agent with respect to the principal functions represented by N.

Theorem 6.1 also tells us that if our network is acyclic and composed of accepting processes then the same is true in the failures model.

The situation with networks where there is no cycle of output requests is not so straightforward, since adding buffers can actually allow such cycles where there were none before. This can occur, for example, because inserting buffering on one of several paths from one process P to another one Q can cause Q to be able to reach control states not reachable without the buffering. Q may then resolve the nondeterminism available to it (in the definition of a functional agent) differently.

However there are some circumstances in which we can guarantee that networks of accepting functional agents will be failures buffer tolerant. For example, since the addition of buffering does not affect whether a functional agent satisfies the preconditions of Proposition 6.2 it follows that we can add arbitrary buffering to such networks without losing the result.

In [22, 18], the idea of *variant functions* is introduced as a tool for proving deadlock absent. The idea is that for each component process of a network we find a function that maps its states into some order (usually the real numbers) such that whenever P has an ungranted request to Q then P's variant is greater than (sometimes weakened to greater than or equal to) Q's. In any cycle of ungranted requests the variant would have to decrease all the way round, so unless they are all equal (if the result permits this) the cycle cannot exist. Importantly for us, the ordering across ungranted requests is judged of the two processes running together as a pair, not in the context of the overall network.

Now consider how this is affected if there is a buffer on the channel. For simplicity we will consider only the case where there is at most one channel between two given processes: let's suppose the channel leads from P to Q. If the request is from $P\diamond>$ to Q then P may have advanced beyond the point where it wanted to output to Q because the output may now be queued in the buffer. P may however be waiting in a subsequent state for the buffer to accept a subsequent output, while the buffer is waiting for Q.

If the request is from Q to $P\diamond>$ then Q is waiting for input from $P\diamond>$. In that case the buffer is certainly empty so this is a combination of states that could have been reached in the binary combination of P and Q without buffering, though not necessarily on the same trace (which the buffering might have affected).

It follows that in any cycle of ungranted requests in $N^\diamond$ each ungranted request over a buffered channel can either occur in the corresponding pair of processes, or the requesting process has gone beyond a point where the ungranted request could have so occurred, and the target of the request has not moved. With a general system of variants this tells us nothing, but if the variants are all non-decreasing (namely, a process communicating never decreases its variant) then we can guarantee the absence of cycles of ungranted requests. This, of course, proves the absence of cycles of output requests. It proves that N is strongly buffer tolerant with respect to the specification of being deadlock free. (This is a general result that does not depend on the nodes being functional agents.)

It is interesting that the examples with nondecreasing variant functions in [22, 18] are all analogues of systolic circuits, namely networks where each node communicates precisely once on each of its channels on every cycle in some fixed order except that they are allowed to perform groups of them in parallel (i.e. offering all members of the set, gradually decreasing until each has happened once). Following [18] (where they were generalised from [3]) we will call these Cyclic Communication Networks, or CCNs.

The results of [3, 22] show that any deadlock-free connected CCN has a non-decreasing system of variant functions. We obtain the following result as a corollary.

PROPOSITION 7.1 *CCNs are strongly buffer tolerant with respect to the specification of being deadlock free.*

Except for nodes with just one input channel, these cannot in principle be accepting, as accepting processes do not constrain the order of their inputs. They may fail to be functional agents with respect to the principal network functions since the order in which internal communication takes place can delay outputs being available beyond where these functions predict. For example if, in the network in Figure 5, the process A cyclically inputs, then outputs to B and then to C; and B inputs externally before inputting from A, then each output from C can be delayed by B waiting for its external input, though this is not reflected in the principal function for C's output.

Such networks are, nevertheless, functional agents with respect to perhaps different functions where each output is computed from all inputs in the same

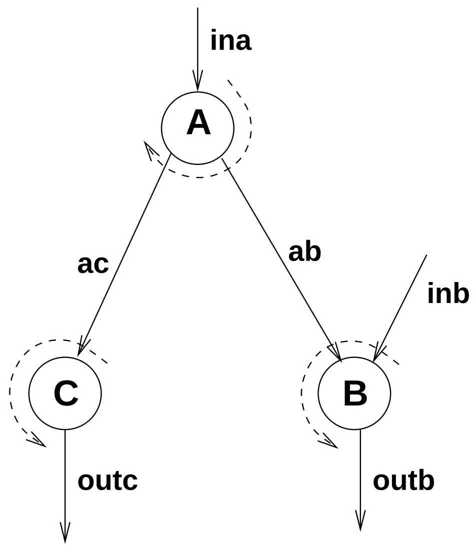

Figure 5: Inputs can delay the functional properties of CCN's

component of the network, not just those from which there is a direct input to output path. So in the above example, B's external input would be one of the streams computing C's output. For each individual output by any of the nodes in our network we can say exactly which overall inputs it depends on:

- All external inputs made by the same node prior to the output, and
- the union of all those sets of inputs which the sending of all its prior internal inputs depended on. (One communication does not precede another if they occur in parallel in a node.)

For a given set of input streams in a deadlock-free CCN, the output streams which are available will be precisely those prefixes of the principal functions such that all the inputs required for these outputs are present in the input streams. So in our example, since the first communication on *outc* depends on the first in *inb* by the above rules, the function associated with *outc* will give the empty sequence unless the input on *inb* is nonempty.

The addition of arbitrary nondeterministic buffering may mean that, while N remains trace functional with respect to the principal functions and deadlock free, it may cease to be a functional agent itself (though outputs can still be guaranteed up to the limits calculated above). For example if arbitrary buffering were placed in our example between A and B, it becomes nondeterministic whether our network can or cannot output on *outc* before *inb*.

8 Confluence

A process or computation is said to be *confluent* if, whenever a state has two alternative actions α and β, then performing either of them does not preclude the other and furthermore performing them in either order leads to equivalent states. The idea of confluence is closely related to the ideas behind the Church-Rosser theorem in λ-calculus and was introduced into process algebra by Milner [15, 16].

In CSP terms any process defined without any choice operator (namely prefix-choice, channel input, external choice, internal choice, time-out and interrupt), the constant *Chaos*, nondeterminism-introducing parallel operators or renamings which are not bijective is naturally confluent. In other words one is allowed to use single-event prefixing, $SKIP$, $STOP$, RUN_A, sequencing, alphabetised parallel ($P\ {}_X\|_Y\ Q$) and hiding, plus recursion. However in the spirit of CSP it is sensible to specify that hiding and recursion do not introduce divergence in order to simplify the theory. We can call this subset of the language *confluent CSP*.

The trace sets of confluent processes satisfy the following law: if $s, t \in traces(P)$ then $s\hat{\ }(t - s) \in traces(P)$. Here $t - s$ is t with elements from s deleted according to mulitplicity, earliest first. Thus $\langle a, c, b, a\rangle - \langle b, a, d\rangle = \langle c, a\rangle$.

A set of traces is said to be confluent if it satisfies this property, which can be proved by induction to be equivalent to the following apparently weaker one:

- If $s\hat{\ }\langle a\rangle\hat{\ }t$ and $s\hat{\ }\langle b\rangle$ (with $a \neq b$) are both in T, so is $s\hat{\ }\langle b, a\rangle\hat{\ }(t - \langle b\rangle)$.

There is a close correspondence between confluent trace sets and Mazurkeiwicz traces [14] in which independence relations are between numbered events (e.g. the 4th a is independent of the 3rd b).

Furthermore, subject to being divergence free, confluent processes are deterministic in the usual CSP sense of not being able either to accept or refuse the same trace. They are thus, like all deterministic processes, completely described by their traces. (Milner makes a similar observation.)

One of the most interesting features of confluent processes is that hiding does not introduce nondeterminism into them, except by the possible introduction of divergence.[9]

It is helpful to liberalise the definition of confluence a little, recognising the difference between communication on a channel and the value that is passed down it. Specifically we will call a value-passing process *channel-confluent* if the result of applying the abstraction which forgets all the data components of channels is confluent in the above sense. In other words its pattern of communication is independent of which data it is sent and is confluent.

Notice that all CCN network components are channel confluent by definition, and hence (thanks to the confluent processes being closed under parallel and hiding) so are networks composed of them. In fact networks of sequential channel-confluent processes are essentially the same as CCN's, if we allow infinitely long "cycles" in the latter!

It has been observed in various places (for example [4, 3]) that networks of different sorts of confluent and deadlock-free processes are themselves deadlock free if there is no cycle of dependencies amongst its communications, where one communication (say $(c, 3)$, the third occurrence of c) depends on another (say $(d, 4)$) if there is a component with both events in its alphabet which forces the fourth d to be before the third c. This is closely related to the observation that was made by Dijkstra and Scholten about sequential CCNs in [3] and that by the author about general CCNs in[22]. Any such cycle-free (channel-)confluent network satisfying one additional condition (see below) will, by the same construction used in [22], have an increasing system of *selective variant* functions in the sense of Deadlock Rule 11 of [18]:

- The transitive closure of the union of the partial orders on numbered

[9] Those familiar with the standard model of cryptographic protocols will recognise that the standard model of the intruder is confluent. They might also recognise that the partial-order reduction operation `chase` which is used to great effect in implementing the intruder is always safe to use on confluent processes (including the results of applying hiding to other confluent processes, where divergence-free) is always semantics-preserving because these processes are always deterministic.

events (a, n) represented by the individual components is (by the absence of cycles) irreflexive and hence a partial order.

- This order can be extended (like all partial orders) to a linear order.
- Like any countable linear order, it can be embedded in the interval $[0, 1]$ (a subset of the real numbers). Let $\rho(a, n)$ be the number associated with the event (a, n).
- If the component process P_i has terminated, give it variant 2. If it has not terminated, its variant is the least $\rho(a, n)$ amongst the events it can do next. It selects the process with which it shares that event.
- Now, whenever process P_i is waiting for process P_j, P_j cannot have terminated thanks to the following additional condition:

  > If a is an event shared by two processes, then if either of them has a finite bound on the number of times it communicates a, then the other has the same bound.

- If neither have terminated and (a, n) is P_i's selected event, then P_j must be waiting for an event that precedes (a, n). Hence P_j's variant is strictly less than P_i's.
- If P_i has terminated then it simply waits for the others to terminate and cannot, by the above, be part of any cycle of ungranted requests.

We can conclude[10] the following by our earlier remarks about variants.

Proposition 8.1 *Every channel-confluent network is strongly buffer tolerant with respect to deadlock freedom.*

It is worth noticing that any fixed-size deterministic buffer is channel confluent, but that any buffer which is either nondeterministic or can ever reduce in capacity is not. (The only way in which the latter could manifest itself in a deterministic process would be a buffer which could accept some input before a particular output but not after it, in direct contradiction to the definition of confluence.)

[10] This result follows straightforwardly from the properties of confluent systems alone if we restrict the scope to the insertion of only confluent buffers on channels. There are, however, many non-confluent buffers.

The above result is similar to one in [13].

The work so far shows how useful the idea of acceptingness can be. Unfortunately it is by no means always a natural property for a functional agent to have. If the role of the function is, for example, to pair off the values that it inputs on two channels, then we can only make our process accepting by giving it an unbounded memory for the excess inputs it might have received on one or other. In this section we seek to generalise it in a way that retains some of its useful properties, especially those that motivate the next section.

What we want to specify is that, within some given programme of inputs which may well allow considerably fewer all orders, the process it obliged to accept in any order. In other words any sequence of inputs consistent with the programme Ξ (a prefix-closed set of sequences of input channels) should be accepted. For reasons which will become apparent in the next section we will want the programme to have the following two properties:

- It must be *exhaustive*, namely the process never accepts a sequence of inputs outside the programme.
- It must be *confluent*, as described above.

DEFINITION 8.1 *We will say that a process P is accepting relative to the input schedule Ξ provided that schedule is exhaustive for P, and whenever (s, O) is a failure, and $ins(s)$ is the sequence of input channels in s, P can refuse no input on any channel c such that $ins(s)\hat{}\langle c\rangle \in \Xi$.*

Note that being accepting is equivalent to being so with respect to the set of all sequences of input channels, and that a deadlock-free process which has a single pre-determined order of inputs is accepting with respect to the schedule of all prefixes of that order.

Any node of a CCN is accepting relative to a schedule which is of this latter form except that any parallel inputs introduce confluent branching within each cycle.

Note that any schedule described as the set of traces of some CSP process formed from the syntax of confluent CSP discussed above is automatically confluent.

The condition of inputting acceptingly with respect to some confluent Ξ is a much weaker condition than acceptingness, since it allows the process to completely prescribe its sequence of inputs. More generally it allows them to input on any channel for which they are "ready" in some sense.

9 Checking functionality and confluence via determinism

In the preceding sections we have seen that functional and confluent processes play natural roles in for buffer tolerant systems. In many cases it will be obvious from how a process is written that it is *meant* to be a functional or confluent, but we should still check that it is. In other cases the property might be almost accidental and emerge naturally as a consequence of aiming for buffer tolerance. In any case it would be useful to have a way of checking to see if a given process is a functional agent.

In the functional case, even if the function itself is known (which it may not be) this is not an easy problem, since the most general functional agent with respect to any given function is almost always infinite state and therefore problematic to check using tools like FDR.

Fortunately we can develop some of the ideas presented in [20] to provide some striking, and importantly finitely checkable, characterisations of these types of process. They are closely related to the way developed there of checking if a process is a buffer. They are especially relevant to the present paper since they both take the form "If P has an appropriate buffer-tolerance property, then P is confluent/deterministic".

9.1 Characterising confluent behaviour

We first remark that if we can check confluence then we can easily also check for channel confluence and being accepting relative to a confluent input schedule. The first of these is as a result of the following straightforward proposition.

PROPOSITION 9.1 *The process P is channel confluent if and only if $P[\![F]\!]$ is confluent, where F is the* forgetful *renaming that removes the data components of all channels.*

The second follows from:

PROPOSITION 9.2 *The FOP process P is accepting relative to some confluent input schedule if and only if $(P \setminus O)[\![F]\!]$ is confluent, where O is its set of outputs and F is as above.*

So we will concentrate on characterising the basic property of confluence. Recall that confluent processes are deterministic, and if they are combined (in a divergence-free way) using constructors from confluent CSP, then they remain confluent.

Suppose P is confluent, that a is one of its events and that a' is an event not used by P. Then

$$CP(a, a') = a \rightarrow a' \rightarrow CP(a, a')$$

is confluent, as is

$$C(a, P) = (P[\![^{a'}/a]\!] \underset{\{a'\}}{\|} C(a, a')) \setminus \{a'\}$$

It follows that $C(a, P)$, in which the communicating of a is given an inwards buffer of size one from the environment to P, is deterministic. This makes quite a subtle observation about the nature of communication by confluent processes.

Just how subtle is demonstrated by the following result.

THEOREM 9.3 *Suppose that either of the following holds:*

(i) $C(a, P)$ *is deterministic for all events* a *of* P.

(ii) $C^*(P) = C(a_1, C(a_2, C(..., C(a_n, P)...)))$ *is deterministic, where* $a_1 \ldots a_n$ *are all the events of* P. *(Note that the value of* $C^(P)$ *does not depend on the order of* a_i*, but that it is important that each event is listed exactly once.)*

Then P *is confluent.*

PROOF Suppose first that condition (i) applies. We will first prove that P is deterministic, and then that its trace set is confluent.

If P were not deterministic, then it would have a trace s after which it could both accept and refuse some event a. After $s\hat{}\langle a\rangle$, $C(a, P)$ can accept a (via the final a of $s\hat{}\langle a\rangle$ going into the buffer, being accepted by P, and a further a being accepted by the buffer) or refuse a (the final a of $s\hat{}\langle a\rangle$ stays in the buffer because P refuses it). So both $C(a, P)$ is then nondeterministic too, in contradiction to our assumption.

We also remark that if P is nondeterministic in the manner above then so is $C(b, P)$ for any $b \neq a$ via an even easier argument. It follows that, in general, if P is nondeterministic then so is any $C(a, P)$. We will use this fact in the proof of (ii) below.

Now suppose (in the proof of (i)) that P does not have a confluent trace set. Then it has traces $s\hat{}\langle a\rangle\langle t\rangle$ and $s\hat{}\langle b\rangle$ for $a \neq b$, but not $s\hat{}\langle b, a\rangle\hat{}(t - \langle b\rangle)$.

Observe that $C(b, P)$ has the trace $s\hat{}\langle b, a\rangle\hat{}(t - \langle b\rangle)$ since the buffer can take in a b at the relevant point and communicate it to P at the point of the first b (if any) of t. (At all other times the buffer just passes b's directly to P at the points in the trace where a b occurs. Let's call this execution α.

It can also communicate the trace $s\hat{}\langle b\rangle$ via the buffer passing all actions directly on to P (i.e. P performs the same trace). Following on from this experiment, we can try to get P to perform the various members of $\langle a\rangle\hat{}(t - \langle b\rangle)$ in turn (inputting b's into the buffer and immediately outputting them). At some point, because $s\hat{}\langle b, a\rangle\hat{}(t - \langle b\rangle)$ is not a trace of P, it will refuse the next event. We can assume that at this point it has performed $u < \langle a\rangle\hat{}(t - \langle b\rangle)$ and that the next event in $\langle a\rangle\hat{}(t - \langle b\rangle)$ is c. Let's call this execution β.

There are two cases to consider. If $c = b$ then we know that it is accepted by the buffer, so β can be extended by a b. However a second b is offered (i.e. a b after $s\hat{}\langle b\rangle\hat{}u\hat{}\langle b\rangle$) it is refused. In α, we know that b is not the first b in t, since that is deleted by $t - \langle b\rangle$, and that therefore the buffer is empty after $s\hat{}\langle b\rangle\hat{}u$. Since P does not, in α, refuse b at this point out it follows that $C(b, P)$ can perform $s\hat{}\langle b\rangle\hat{}u\hat{}\langle b, b\rangle$ contradicting the determinism of this process.

If $c \neq b$ then since the buffer never accepts c it follows that $C(b, P)$ can also (in β) refuse c after $s\hat{}\langle b\rangle\hat{}u$. In α we know that c will be accepted after the same trace, again menaing that $C(b, P)$ is nondeterministic.

So we conclude that condition (i) is indeed sufficient to prove confluence.

For condition (ii), we know that if P is confluent then $C^*(P)$ is deterministic: note that the hiding involved here can never introduce divergence, and the same holds for $C(a, P)$. If P is not confluent we know that $C(b, P)$ is nondeterministic for some b. It follows (by the remark above that $C(a, \cdot)$ preserves nondeterminism) that applying all the other $C(a, \cdot)$ operators required to build $C^*(P)$ creates a nondeterministic process. So (ii) is proved also. ∎

What the above result shows is that confluent processes are completely characterised by deterministic under a particular sort of transformation. The version of the result with $C^*(P)$ demonstrates that there is a *single* transformation which captures the property. This result has obvious relationships with the characterisation of noninterference in [23, 17, 18] (namely the absence of information flow from a user of P with alphabet H to one with alphabet L) as the determinism of $\mathcal{L}_H(P)$, the lazy abstraction of P.

Since $C^*(P)$ will have more states than any particular $C(a, P)$, it is not clear which of the two criteria in the above result will be more efficient to check on FDR.

We now give an alternative method of checking for confluence which is based on the idea of [19]. It is less elegant but probably more efficient.

A divergence-free and deterministic process P fails to be confluent if and only if there are traces $s\hat{\ }\langle b\rangle$ and $s\hat{\ }t$ (where t may be empty or begin with b) such that, after $s\hat{\ }\langle b\rangle$, P will not refuse any of the events of $t - \langle b\rangle$ presented in order.

Therefore we can test for confluence by checking for determinism of P and running two copies of P in parallel, synchronising for any number of initial events and with the ability, at any time, to fix a communication b of one of the copies (P_1, say), and observe all communications of P_2 which are then, apart from the first b, passed on to P_1. If this passing-on fails at any stage (by deadlock) then P is not confluent; otherwise it is. All of this can be done by double renaming all the events of P_1 and P_2 to two events each (one copy in common, the others disjoint between P_1 and P_2), and then putting their parallel combination in parallel with a suitable regulator.

9.2 Characterising functional behaviour

It is possible to get a similar characterisation for functional agents, though the proof is harder. The first thing we do is to define an abstract property which is something that all functional agents always have.

DEFINITION 9.1 *Suppose the alphabet of the divergence-free process P is $O \cup I$, where O represents outputs and I inputs, each consisting of the events associated with one or more channels. Then P is* output deterministic (OD) *provided, for all $s \in traces(P)$ and $c.x, c.y \in O$:*

(i) If $s\hat{\ }\langle c.x\rangle \in traces(P)$ then $(s, O) \notin failures(P)$.

(ii) If $s\hat{}\langle c.x\rangle$, $s\hat{}\langle c.y\rangle \in \mathit{traces}(P)$ *then* $x = y$.

In other words if an output is possible then (i) the process cannot refuse all outputs and (ii) any other output on the same channel must be the same.

Note that this is a slightly changed definition from that used in [20]: that one was aimed at processes with a single output channel, and only a form of (ii) applied there. Note that the above definition still allows a considerable amount of nondeterminism: not only whether inputs are accepted but also over which channel the process outputs on.

DEFINITION 9.2 *If* $s \in \mathit{traces}(P)$, *we say that* s *is* output maximal *if, for all* $t \leq s$, *such that either* $t = s$ *or there is* $i \in I$ *with* $t\hat{}\langle i\rangle \leq s$, *we have* $(t, O) \in \mathit{failures}(P)$.

If $u\hat{}s \in \mathit{traces}(P)$, *we say that* $u\hat{}s$ *is* output maximal after u *if, for all* $t \leq s$, *such that either* $t = s$ *or there is* $i \in I$ *with* $t\hat{}\langle i\rangle \leq s$, *we have* $(u\hat{}t, O) \in \mathit{failures}(P)$.

Thus a trace is output maximal if inputs only occur when all outputs are exhausted, and also outputs are exhausted at the end of the trace.

LEMMA 9.4 *Suppose* P *is accepting relative to* Ξ *and FOP, that* $s \in \mathit{traces}(P)$ *and* ι *is a sequence of input events such that* $\mathit{chans}(s\hat{}\iota)$ *belongs to* Ξ. *Then there is a maximal trace* $t \in (O \cup I)^+$ *such that (i)* $s\hat{}t$ *is output maximal after* s *and (ii)* $t \upharpoonright I = \iota$. *(Here,* $\mathit{chans}(s)$ *returns the sequence of channels of the events in* s.*)*

If P *is OD and has only one output channel then that trace* t *is unique.*

PROOF We build $s\hat{}t$ a step at a time starting from a state on which P has done s. At each point one of the following three holds

- An output event o is possible. If so, add it to t.
- O is a refusal after the current trace $s\hat{}t$ and ι is non-empty. By acceptingness the trace $s\hat{}t\hat{}\langle i\rangle$ belongs to P, where i is the first member of ι. If so add i to t and delete it from ι.
- $(s\hat{}t, O) \in \mathit{failures}(P)$ and $\iota = \langle\rangle$. If so we are finished.

Clearly this procedure is guaranteed to terminate by FOP and generates an output maximal trace after s.

If P has only one output channel and is OD then (i) if there is an output event possible it is unique and (ii) the output event is only available when $(s\hat{\ }t, O) \notin \mathit{failures}(P)$. It follows that at every stage there is no choice over how to carry out the above procedure. ∎

We showed earlier how to add arbitrary buffering (meaning length 0 or more, with nondeterministically varying length) onto any channel of a network. That definition can trivially be adapted to add a *0/1-buffer* which varies nondeterministically between lengths 0 and 1 onto a channel. Unlike the general construction, this modified one is finitary.

DEFINITION 9.3 *If P is any process, let $\Delta(P)$ be P with a 0/1-buffer on each input channel and which is constrained so that no more than input item in total is buffered during its entire history. This, in some sense, represents the smallest possible non-trivial amount of input buffering for P*

THEOREM 9.5 *Suppose P is a process which is FOP and is accepting with respect to the confluent schedule Ξ. Then the following are equivalent.*

(1) $\Delta(P) \setminus (O \setminus \{\mid d \mid\})$ is OD for all $d \in \mathit{Outchans}(P)$.

(2) $\mathit{buff}(P) \setminus (O \setminus \{\mid d \mid\})$ (and hence all its refinements) are OD for every $d \in \mathit{Outchans}(()P)$.

(3) $\mathit{buff}^{\infty}(P)$ is deterministic.

(4) P is a functional agent.

PROOF The result will be proved if we can show (2) $\Rightarrow$ (1), (3) $\Rightarrow$ (2), (4) $\Rightarrow$ (3), and (1) $\Rightarrow$ (4).

(2) $\Rightarrow$ (1) is trivial since $\Delta(P) \setminus (O \setminus \{\mid d \mid\}) \sqsupseteq \mathit{buff}P \setminus (O \setminus \{\mid d \mid\})$.

It is interesting to note that the $\Delta(P) \setminus (O \setminus \{\mid d \mid\})$ all being OD is a strictly stronger statement that saying the same thing without the hiding. For an example, consider the process

$$(a!1 \to b!2 \to STOP) \,\Box\, (b!1 \to a!2 \to STOP)$$

This is not functional but is output deterministic in each of its two output channels. However hiding either channel means it is no longer output deterministic.

We now turn to (3) $\Rightarrow$ (2). We will show that if one of the processes of (2) fails to be OD then that of (3) fails to be deterministic. Suppose the relevant output channel is d. Observe that $\mathit{buff}^{\infty}(P)$ has all the traces of $\mathit{buff}(P) \setminus (O \setminus \{\mid d \mid\})$ since all outputs on other channels can be absorbed by the output buffers on those channels. The output buffer of d will always contain just what the one in $\mathit{buff}(P) \setminus (O \setminus \{\mid d \mid\})$ does. Furthermore, if $\mathit{buff}(P) \setminus (O \setminus \{\mid d \mid\})$ is stable after trace s, then so can be $\mathit{buff}^{\infty}(()P)$ with P being in the same state. For, thanks to the hiding, no output can be available to any of the non-d output buffers in a state where $\mathit{buff}(P) \setminus (O \setminus \{\mid d \mid\})$ is stable.

$\mathit{buff}(P) \setminus (O \setminus \{\mid d \mid\})$ can fail to be OD after some trace s in two ways:

- Both $s\hat{}\langle d.x\rangle$ and $s\hat{}\langle d.y\rangle$ may be traces for $x \neq y$ and hence also traces of $\mathit{buff}^{\infty}(P)$. In this case, since the events $d.x$ and $d.y$ are both offered in the latter process by an output buffer which never offers a choice of output, it follows that $\mathit{buff}^{\infty}(P)$ is nondeterministic.
- (s, O) may be a failure of $\mathit{buff}(P) \setminus (O \setminus \{\mid d \mid\})$ and $s\hat{}\langle d.x\rangle$ a trace. In the state where the failure occurs we know that the d output buffer of $\mathit{buff}(P) \setminus (O \setminus \{\mid d \mid\})$ is empty and that P refuses all outputs. It follows that the same two things are true when the same stable state is reached in $\mathit{buff}^{\infty}(P)$ after s. Hence the latter process refuses all outputs on d (though maybe not others). But this process can also communicate $s\hat{}\langle d.x\rangle$, meaning it is nondeterministic.

So (3) $\Rightarrow$ (2) is proved.

Now suppose that $\mathit{buff}^{\infty}(P)$ is nondeterminstic. To establish (4) $\Rightarrow$ (3) we will show that P is not a functional agent. Suppose for contradiction that it is and the function predicting output on each channel d is f_d. By construction the nondeterminism in $\mathit{buff}^{\infty}(P)$ cannot arise from it refusing an input. It follows that there is some output $d.x$ which, after trace s, $\mathit{buff}^{\infty}(P)$ can both accept and refuse. There are two possibilities to consider: either it refuses $d.x$ because it is offering a different value $d.y$ on the same channel, or it refuses $d.x$ because P is refusing every output (noting that if P

offered anything the buffers would absorb it, creating a τ), and every input from available from input buffers that are consistent with Ξ.

If the first of these two possibilities had occurred then P will have absorbed trace is_1 of inputs to generate $d.x$ and is_2 of inputs to generate $d.y$. (In both cases the sequence of input values in is_1 and is_2 on a given channel c are prefixes of the sequence of values on c in s.) It is, however, a property of confluent schedules that there are extensions is_1' and is_2' of is_1 and is_2 which contain the same values on every input channel. The acceptingness and FOP of P means that these extensions can respectively be accepted from the points at which is_1 and is_2 have been. Necessarily the sequences of values predicted by f_d from the inputs is_1' and is_2' are the same, but this contradicts the fact that after is_1 and is_2 P had produced incomparable output sequences along d.

The second route to nondeterminism cannot occur either, for similar reasons. Necessarily the functional agent P will have absorbed more inputs on at least one input channel c than it did in refusing all outputs. But in the second case the extra input on each such c is available in the relevant buffer to P, and properties of confluent schedules imply that at least one of these must be accepted by P, giving a contradiction to the fact that $\mathit{buff}^\infty(P)$ is stable when it refuses O. We can conclude that (4) $\Rightarrow$ (3).

It remains to prove (1) $\rightarrow$ (4). So suppose P satisfies the conditions set out in (1). P is itself OD: if it were not then it would fail on some output channel d. It is easy to see that then $P \setminus (\Sigma \setminus \{| d |\})$ would not be OD, and that this process refines $\Delta(P) \setminus (O \setminus \{| d |\})$.

Lemma 9.4 then allows us to build the functions f_d which we will show define P's functional behaviour. Plainly there is only a need to define f_d for those sets of inputs represented by a trace in Ξ: for any other the value of f_d is calculated from a trace within Ξ that maximises inputs along each channel. We know that the channel sequences are then uniquely defined.

We first define the f_d relative to the traces of inputs consistent with Ξ. In other words we will create f_d as a function from these traces to the output stream on d. We will later show that they only depend on the sequences of value on the input channels.

So suppose $v = \langle i_1, \ldots, i_n \rangle$ is a sequence of inputs consistent with Ξ (i.e. the sequence of channel names is in Ξ.) $f_d(v)$ then equals $t \downarrow d$ where t is the unique output maximal trace produced by Lemma ?? relative to input sequence v starting from the empty trace.

Suppose there were two input trace s and t in which all the individual channel sequences were the same but for which $f_d(t)$ and $f_d(s)$, as defined above, are different. We may assume that s and t (necessarily the same length) are as short as possible and furthermore have, subject to this, as long a common prefix as possible. We can write $s = u\hat{}\langle a\rangle\hat{}v$ and $t = u\hat{}\langle b\rangle\hat{}v'$ for some $a \neq b$. Observe that $s' = u\hat{}\langle b, a\rangle\hat{}v'$ is consistent with Ξ where $v' = v - \langle b\rangle$, and so is $s'' = u\hat{}\langle a, b\rangle\hat{}v'$ by applying the definition of confluence to $s\hat{}\langle a\rangle$ and s'. The fact that s' and s'' agree with t and s for as long as they do implies that $f_d(s') = f_d(t)$ and $f_d(s'') = f_d(s)$.

Now observe that if we run $\Delta(P) \setminus (O \setminus \{\mid d \mid\})$ with the inputs s' then it can produce either $f_d(t)$ or $f_d(s)$ as outputs, with the inputs of b then a being consecutive events. The first is because the buffer may choose to pass all inputs directly to P, which executes exactly the sequence of behaviours it does in computing $f_d(s')$. The second is because the buffer might hold the value b until immediately after P inputs a, so that the trace P executes is the same as in calculating $f_d(s'')$.

In these two behaviours, the OD process $\Delta(P) \setminus (O \setminus \{\mid d \mid\})$ has exactly the same trace until the input of b then a in these two behaviours, and after that the behaviours are output maximal driven by the rest of the inputs in s': the output maximal continuation of the trace $u\hat{}\langle b, a\rangle$ is not unique. this contradicts Lemma ??, proving that the f_ds do indeed depend on only the various input sequences, not the order of their interleaving.

In order to show that P is functional we need show that after a general trace s,

- P can only output values $d.x$ which are predicted by the corresponding f_d, and
- can only refuse to output when all the outputs predicted by all the f_d's have appeared.

It is sufficient to prove this for an arbitrary $P \setminus (O \setminus \{\mid d \mid\})$, since if ever P can refuse to output when there is output pending on some channel d then this hidden process has the same property. So we only have to address this case.

It is easily shown to be equivalent the statement (*) that after every trace s, if no further inputs are offered, $P \setminus (O \setminus \{\mid d \mid\})$, is prepared to output precisely the balance of $f_d(s)$.

Again, if there were a trace s after which the above failed for $P \setminus (O \setminus \{| d |\})$, there would be one with as few as possible events in before an output-maximal tail. In other words, $s = u\hat{}\langle a\rangle\hat{}v$ where a is the last input made before it is forced (i.e. there is an output $d.x$ available at the same point), and subject to this u is as short as possible. Let w be the unique output maximal extension of this trace for no further inputs: so $u\hat{}\langle a\rangle\hat{}v\hat{}w$ is output maximal beyond the a.

We can construct the unique output maximal trace of P starting from u based on the inputs in $\langle a\rangle\hat{}v$, and it will take the form $u\hat{}\langle d.x\rangle\hat{}t$. As earlier, we observe that after $\Delta(P) \setminus (O \setminus \{| d |\})$ has performed u (with nothing ever buffered) and then a, and then an output maximal trace from there on:

- It might perform $v\hat{}w$ because the buffer may pass a straight to P.
- It might perform $\langle d.x\rangle\hat{}(t - \langle a\rangle)$ because the buffer might hold a until the point where it is input in t.

Uniqueness of output-maximal traces tells these two are the same.

Note that in performing $u\hat{}\langle d.x\rangle\hat{}t$, $\Delta(P) \setminus (O \setminus \{| d |\})$ can only perform unforced inputs within u, and therefore its non output-maximal prefix is too short for it to be a counter-example to (*). It follows that the outputs of $\langle d.x\rangle\hat{}t$ are precisely the balance of $f_d(s)$; but we have already shown that these are precisely the outputs of $v\hat{}w$, contradicting our assumption that it is a counter-example to (*). Therefore (*), and (1) $\Rightarrow$ (4), are proved.

The great virtue of alternative (1) in the above theorem is that if P is finite-state then this criterion can be checked finitely using a tool like FDR. Of course the minimalist buffering used should help limit the state space explored. An FDR check for the property of being output deterministic follows essentially the lines set out in [20], which itself followed the check for determinism in [12]. The following works for the case we need, namely with a single output channel, but at the expense of a little more complexity it can be adapted to the multiple case. We simply run two copies of P in parallel, synchronising on inputs and not outputs, but forcing the two copies to make the same output and deadlocking otherwise. To help in this the outputs o of one of the P's are renamed injectively to disjoint events o'. The property is then equivalent to this combination refining the most nondeterministic process than never deadlocks after the first of a pair of outputs: if

$$\begin{aligned} Spec &= STOP \sqcap i?I \to Spec \\ &\quad \sqcap o?O \to o' \to Spec \end{aligned}$$

$$\begin{aligned} Imp &= (P \underset{I}{\parallel} P[\![{}^{o'}/o \mid o \in O]\!]) \underset{O \cup O'}{\parallel} Reg \\ Reg &= o?O \to o' \to Reg \end{aligned}$$

then P being OD is equivalent to *Imp* failures refining *Spec*.

9.3 Applications

It is obvious that we can use the characterisations of confluence and functional behaviour above to check if a given process has one of these properties.

We now have the ability to tell if a given process (accepting wrt some confluent Ξ) is functional without knowing the function(s) f_d in advance. Of course we may very well want to prove that some P is functional with respect to a given set of f_d. For example we can prove a process with one input and one output channel is a buffer by showing it is functional with respect to the identity function.

For any confluent schedule Ξ and any single function f_d on the relevant input streams, we can build the *urgent* implementation of f_d, parameterised by the various input streams, the sequence of input channels to date and the output stream to date:

$$\begin{aligned} & U(f_d, is_1, \ldots, is_n, i, o) \; = \\ &\qquad d!(head(f_d(is_1, \ldots, is_n) - o)) \to \\ &\qquad\qquad U(f_d, is_1, \ldots, is_n, o\hat{\ }\langle head(f_d(is_1, \ldots, is_n) - o)\rangle) \\ &\qquad \lessdot f_d(is_1, \ldots, is_n) \neq o \gtrdot \\ &\qquad \Box\{c_j?x \to U(f_d, is_1, \ldots, is_j\hat{\ }\langle x\rangle, \ldots, is_n, i\hat{\ }\langle c\rangle, o) \mid i\hat{\ }\langle c\rangle \in \Xi\} \end{aligned}$$

This is always a deterministic process, and for many functions it can be simplified considerably. For example, for the identity function of one channel, the initial state where all the sequence parameters are empty is equivalent to $COPY$.

If P is FOP, accepting wrt Ξ and functional with a single output channel d with function f_d then

$$U'(f_d) = U'(f_d) \underset{\Sigma}{\parallel} P$$

where $U'(f_d) = U(f_d, \langle\rangle, \ldots, \langle\rangle, \langle\rangle, \langle\rangle)$. This is because P must always be prepared to communicate everything $U^*(f_d)$ can on all traces allowed by the latter.

We can extend this to a check on the function of P by extending $U'(f)$ by allowing outputs when they are not wanted by f:

$$
\begin{array}{l}
U^+(f_d, is_1, \ldots, is_n, i, o) \;=\; \\
\qquad d!(\mathit{head}(f_d(is_1, \ldots, is_n) - o)) \rightarrow \\
\qquad\qquad U^+(f_d, is_1, \ldots, is_n, o\hat{\ }\langle \mathit{head}(f_d(is_1, \ldots, is_n) - o)\rangle) \\
\qquad \not< f_d(is_1, \ldots, is_n) \neq o \not> \\
\qquad (\Box\{c_j?x \rightarrow U^+(f_d, is_1, \ldots, is_j\hat{\ }\langle x\rangle, \ldots, is_n, i\hat{\ }\langle c\rangle, o) \mid i\hat{\ }\langle c\rangle \in \Xi\} \\
\qquad \Box\; d?x \rightarrow \mathit{STOP})
\end{array}
$$

$$U^*(f_d) \;=\; U^+(f_d, \langle\rangle, \ldots, \langle\rangle, \langle\rangle, \langle\rangle)$$

We then get the following proposition, because every value that P outputs is then known to be one consistent with f_d. The extra option in $U^*(f_d)$ means that on the right hand side P is allowed to produce extra outputs at any stage that are beyond those called for by f_d. If the two sides are equal we know that such outputs never appear.

PROPOSITION 9.6 *Suppose P is functional, FOP, accepting with respect to confluent Ξ and has the single output channel d. Then the function of P is f if and only if*

$$U'(f) = U^*(f) \underset{\Sigma}{\|} P$$

We can check the functions of a functional agent with multiple output channels by using the above on all the processes $P \setminus (O \setminus \{\mid d \mid\})$ for $d \in \mathit{Outchans}(P)$.

So if we define

$$\mathit{COPY}^* = (in?x \rightarrow out!x \rightarrow \mathit{COPY}) \Box (out?x \rightarrow \mathit{STOP})$$

we now know that in general a process P is a buffer if any only if both the following hold:

- $\Delta(P)$ is output deterministic.

- $COPY = COPY^* \underset{\Sigma}{\parallel} P$.

This is a rather better finitary test for being a buffer than that in [20], though it is related.

10 Conclusions

We have shown that there is a deep relationship between buffer tolerance and functional agents. They are natural target specifications for weakly buffer tolerant networks since if we can show (for such a network that N refines some functional agent, then $N^\diamond$ refines the most nondeterministic functional agent for the same set of functions. We showed that networks of these agents have a number of natural buffer tolerance properties, and finally showed that they are essentially the only processes that are tolerant, with respect to the property of being output deterministic, of the addition of input buffering. This shows these are the only specifications sufficiently strong to specify *when* output can appear and *what* value will appear on each channel, which we can hope to prove of weakly (or leftwards) buffer tolerant systems.

We have also shown how deep inductive proofs can be used to demonstrate a finitary characterisation of functional agents. These results bear comparison with the characterisations of noninterference given in [23, 17, 18], where it is shown that deciding if a process can pass information between users can be reduced to checking the determinism of a modified process. In that case we check the determinism of the process with the most general model of a high-level user in order to see if the nondeterminism it creates can be visible to a low level one. In this paper the check is that the addition of arbitrary external buffering does not create nondeterminism in the relationship between inputs and outputs.

The latter is obviously an appealing concept in the world of buffer tolerance, so the fact that it characterises functional agents simply emphasises the importance of these.

The finitary check provided by Theorem 9.5 will be useful if checking that a sequential component process, created in a language like CSP which does not guarantee functional behaviour, actually has it. It will also be useful in checking that a parallel composition of processes also has this property.

Recall that we showed how, if a network N had been proved deadlock free using nondecreasing variant functions, then $N^\diamond$ is also deadlock free. It will be interesting to see if this has applications outside the world of CCNs.

11 Conclusions

In Part 1 of this paper we studied the definition of buffer tolerance and discovered a variety of conditions on processes that are useful for reasoning about it. We have a general result – Theorem 2.6 – about constructing buffer tolerant networks and seen that it can naturally be applied to chains and certain forms of tree network.

Theorem 2.6 requires rightward or leftward buffer tolerance. Rightward buffer tolerance is not something we should expect to be true for the generality of processes with more than one input channel since having buffers on the left will tend to make for a wider range of orders of inputs than when they have been moved to the right. (This is not true of accepting processes.) The same is true of processes with more than one output channel for leftward buffer tolerance.

We showed in Part 2 that there is a deep relationship between buffer tolerance, functional agents and confluence. Functional agents are natural target specifications for weakly buffer tolerant networks since if we can show (for such a network that N refines some functional agent, then $N^\diamond$ refines the most nondeterministic functional agent for the same set of functions. We showed that networks of these agents have a number of natural buffer tolerance properties, and finally showed that they are essentially the only processes that are tolerant, with respect to the property of being output deterministic, of the addition of input buffering. This shows these are the only specifications sufficiently strong to specify *when* output can appear and *what* value will appear on each channel, which we can hope to prove of weakly (or leftwards) buffer tolerant systems.

We also showed how Milner's concept of confluence was related to buffer tolerance, both because confluent systems have natural buffer tolerance properties and because the idea allows us to take proper control of the idea of functional agent.

As will be apparent from this paper's bibliography, authors from a number of communities have noticed aspects of the relationships between functional behaviour and confluence with buffering and deadlock freedom. The author has no doubt that there is more literature that he has not discovered. What we have sought to do here is to systematise these ideas with the specific objective of buffer tolerance in mind, and to bring it all firmly within the process algebra framework.

We have also shown how buffering ideas can used to demonstrate a finitary characterisations of confluence and of functional agents. These results bear comparison with the characterisations of noninterference given in [23, 17, 18], where it is shown that deciding if a process can pass information between users can be reduced to checking the determinism of a modified process. In that case we check the determinism of the process with the most general model of a high-level user in order to see if the nondeterminism it creates can be visible to a low level one. Both of our checks also took the form of showning that placing the process under examination into an appropriate context yields a deterministic (selectively so in the function case) process.

The failures divergences model is known [21] to be fully abstract with respect to deciding if a process is deterministic. Therefore the results alluded to in the previous paragraph, as well as the natural specification that model provides of a buffer, indicate that it is a very natural home for reasoning about buffer tolerance, and functional and confluent behaviour viewed in an extensional as opposed to intensional way.

The fact that CSP models have a theory of refinement was important in the way we defined buffer tolerance, but it may well be possible to reformulate it without: for example weak buffer tolerance might become $N^{\diamond\diamond} \cong \mathit{buff}(N)$.

There is almost certainly a lot more to discover about buffer tolerance, in particular about the ways and pragmatics of using it in practice. This will have to be the subject of future work.

Appendix: Notation

This paper follows the notation of [18], from which most of the following is taken.

$\mathbb{N}$	natural numbers ($\{0, 1, 2, \ldots\}$)
Σ	(Sigma): alphabet of all communications
τ	(tau): the invisible action
Σ^τ	$\Sigma \cup \{\tau\}$
A^*	set of all finite sequences over A
$\langle\rangle$	the empty sequence
$\langle a_1, \ldots, a_n\rangle$	the sequence containing a_1,..., a_n in that order
a^ω	the infinite trace $\langle a, a, a, \ldots\rangle$
$s\hat{\ }t$	concatenation of two sequences
$s \setminus X$	hiding: all members of X deleted from s
$s \leq t$	($\equiv \exists\, u.s\hat{\ }u = t$) prefix order

Processes:

$\mu\, p.P$	recursion
$a \rightarrow P$	prefixing
$?x : A \rightarrow P$	prefix choice
$P \;\Box\; Q$	external choice
$P \sqcap Q, \quad \bigsqcap S$	nondeterministic choice
$P \underset{X}{\parallel} Q$	generalised parallel
$P \setminus X$	hiding
$P[\![R]\!]$	renaming (relational)
$P \gg Q$	chaining
$P \triangleright Q$	"time-out" operator (sliding choice)
$P[x/y]$	substitution (for a free identifier x)

Transition Systems:

$P \xrightarrow{a} Q$	$(a \in \Sigma \cup \{\tau\})$ single action transition
$P \overset{s}{\Longrightarrow} Q$	$(s \in \Sigma^*)$ multiple action transition with τ's removed
$P \overset{t}{\longmapsto} Q$	$(t \in (\Sigma^\tau))^*)$ multiple action transition with τ's retained

Buffering:

$COPY$	deterministic one-place buffer
$BUFF_{\langle\rangle}$	the most nondeterministic buffer process
$ZB_{\langle\rangle}$	the most nondeterministic zero-buffer process
$Inchans(P)$	input channels of P
$Outchans(P)$	output channels of P
$P\diamond c>$	adds arbitrary buffering to output channel c
$\diamond c> P$	adds arbitrary buffering to input channel c
$P\diamond c>$	adds arbitrary buffering to unique output channel
$\diamond c> P$	adds arbitrary buffering to unique input channel
$P\infty c>$ (etc)	adds unbounded buffer to (output channel c)
$P >\!\!\diamond\!\!> Q$	adds arbitrary buffering to chain operator
$inbuff(P)$	adds arbitrary buffering to all input channels
$outbuff(P)$	adds arbitrary buffering to all output channels
$buffP$	adds arbitrary buffering to all (external) channels
$N^{\diamond C}$	adds arbitrary buffering to all channels in C
$N^{\diamond}$	adds arbitrary buffering to all internal channels C
$N^{\diamond\diamond}$	adds arbitrary buffering to all internal and external channels C

Acknowledgements

This work was inspired by Philippa Hopcroft and Guy Broadfoot who persuaded me it was needed from a practical standpoint, and by Mark Joseph's lecture on asynchronous CSP at the *25 years of CSP* meeting in July 2004. Both he and Samson Abramsky helped me out with pointers to related work. It was funded by a grant from US ONR.

References

[1] B. Boigelot, P Godefroid, B.Willems and P. Wolper, *The power of QDDs*, Proceedings of SAS 1997.

[2] B. Boigelot and P. Wolper, *Verifying systems with infinite but regular state spaces*, Proceedings of CAV 1998.

[3] E.W. Dijkstra and C.S. Scholten, *A class of simple communication patterns* EWD643 in 'Selected writings on computing', Springer-Verlag 1982.

[4] Formal Systems (Europe) Ltd, *FDR2 user manual and tutorial*, `www.fsel.com/documentation/fdr2/html`.

[5] M.C.W. Geilen and T. Basten, *Requirements on the execution of Kahn process networks*, Proc. ESOP 2003, LNCS 2618

[6] P. Godefroid and D.E. Long, *Symbolic Protocol Verification with Queue BDDs* Formal Methods in System Design **14**, 3, pp257-271, 1999.

[7] M.H. Goldsmith and A.W. Roscoe, *Transforming occam programs*, in The Designs and Application of Parallel Digital Processors, IEE Conference Publication 298, 1988.

[8] C.A.R. Hoare, *Communicating Sequential Processes*, Prentice-Hall 1985.

[9] M.B. Josephs, *Receptive Process Theory*, Acta Informatica **29**, pp 17–31 (1992).

[10] M.B. Josephs, C.A.R Hoare and He Jifeng, *A theory of asynchronous processes*, PRG-TR-6-89, Oxford University Computing Laboratory 1989.

[11] G. Kahn, *The semantics of a simple language for parallel programming*, Information Processing '74, pp 471–475, North-Holland 1974.

[12] G. Kahn and D.B. MacQueen, *Coroutines and networks of parallel processes*, Information Processing '77, pp 993–998, North-Holland 1977.

[13] R.S. Lazić, *A semantic study of data independence with applications to model checking*, Oxford University D.Phil thesis 1999.

[14] R. Manohar and A. Martin, *Slack elasticity in concurrent computing*, Proc. MPC '98, LNCS 1422, 1998.

[15] A. Mazurkiewicz, *Concurrent program schemes and their interpretation*, TR DAIMI PB-78, Aarhus University 1977,

[16] R. Milner, *A calculus of communicating systems*, LNCS 92, 1980.

[17] R. Milner, *Communication and concurrency*, Prentice-Hall 1989.

[18] A.W. Roscoe, *CSP and determinism in security modelling*, Proc. IEEE Symposium on Security and Privacy, IEEE Computer Society Press, '995.

[19] A.W. Roscoe, *The theory and practice of concurrency*, Prentice-Hall, 1998.

[20] A.W. Roscoe, *On the expressiveness of CSP refinement checking*, To appear in FAC (special issue related to AVOCS '03).

[21] A.W. Roscoe, *Finitary refinement checks for infinitary specifications*, Proceedings of CPA 2004 (IOS Press).

[22] A.W. Roscoe, *Revivals, stuckness and responsiveness*, available from `www.comlab.ox.ac.uk/oucl/work/bill.roscoe/pubs.html`.

[23] A.W. Roscoe and N. Dathi, *The pursuit of deadlock freedom*, Information and Computation **75**, 3, pp 289-337 (1987).

[24] A.W. Roscoe, J.C.P. Woodcock and L. Wulf, *Non-interference through determinism*, JCS **4**, 1, pp 27–54, 1996. (Revised from LNCS 875, Proc ESORICS 94).