Communicating Process Architectures – 2004
Jeremy Martin et al (Eds.)
IOS Press, 2004

Finitary refinement checks for infinitary specifications

A.W. Roscoe
Oxford University Computing Laboratory, Wolfson Building, Parks Road, Oxford OX1 3QD, UK

Abstract. We see how refinement against a variety of infinite-state CSP specifications can be translated into finitary refinement checks. Methods used include turning a process into its own specification inductively, and we recall Wolper's discovery that data independence can be used for this purpose.

1 Introduction

Thanks to the refinement checker FDR[1] [1], the question of what predicates on CSP models are decidable via a refinement check is practically as well as intellectually important. By this I mean: what predicates of a process P are equivalent to the question

$$F(P) \sqsubseteq G(P) \qquad (\dagger)$$

for CSP-definable functions F and G? In [7] I gave fairly complete characterisations of the unexpectedly wide range of predicates that can be represented like this. In particular, for the failures/divergences model over a finite alphabet, the metric/topological closed subsets are precisely the predicates that can be represented by metric-continuous F and G.

However, as observed in [7] this answer is of little practical use since the CSP contexts F and G it generates are, in general, infinitary – indeed, usually, they are in a strong sense infinitely large pieces of syntax. Nevertheless, experience and recent work [5] has shown that many useful and subtle properties can be expressed in practical ways.

In the present paper I will begin to study what can be done within the terms of what is practically possible on FDR. Namely, we will restrict ourselves to F and G which, when applied to a finite-state process P, have $F(P)$ and $G(P)$ finite state also. Note that this definition of a *finitary* function is grounded in the operational semantics of CSP, since that is where the notion of a 'finite state process' naturally rests.

By and large I will concentrate more on providing a range of useful recipes for representing predicates rather than full characterisations, simply because at present I have more of the former than the latter! Also the majority of this initial study will be devoted to discovering how to replace infinite-state fixed specification processes *Spec* for simple checks $Spec \sqsubseteq P$ by finitary checks.

We will find that data independence plays an unexpected large role in this work.

In the next section we will study how to characterise a counter process, and in the following one we will extend this to the related specifications of bag, buffer and stack. We then discuss the potential for moving from our examples to general results.

[1]The references in this paper to FDR's capabilities are based on the tool in May 2004.

2 Counter processes

Anyone who has studied CSP will have encountered infinite-state counter processes with events *up*, *down* and perhaps *iszero*, such as $Count_0$, $Count'_0$, *Zero* and *ZERO*, where

$$\begin{aligned}
Count_0 &= up \rightarrow Count_1 \\
Count_{n+1} &= up \rightarrow Count_{n+2} \\
&\quad \Box\ down \rightarrow Count_n \\
Count'_0 &= up \rightarrow Count'_1 \\
&\quad \Box\ iszero \rightarrow Count'_0 \\
Count'_{n+1} &= up \rightarrow Count'_{n+2} \\
&\quad \Box\ down \rightarrow Count'_n \\
Zero &= up \rightarrow Pos;\ Zero \\
&\quad \Box\ iszero \rightarrow Zero \\
Pos &= up \rightarrow Pos;\ Pos \\
&\quad \Box\ down \rightarrow SKIP \\
ZERO &= up \rightarrow (ZERO \ |||\ down \rightarrow STOP)
\end{aligned}$$

By analogy with the usual CSP specification of a buffer ([6], for example) we might reasonably specify a counter (of the first sort) to be a process which must accept *up* when it has value 0 and *down* when non-zero, and which never accepts more *down*s and *up*s. If (t, X) is a typical failure, this formalises to

(i) $t \in \{up, down^*\} \wedge t \downarrow down \leq t \downarrow up$

(ii) $t \downarrow up = t \downarrow down \Longrightarrow up \notin X$

(iii) $t \downarrow up > t \downarrow down \Longrightarrow down \notin X$

A counter of the second sort must accept *iszero* when zero and never when non-zero. (The value of a counter is defined to be the number of *up*s minus the number of *down*s.) This can be formalised by easy modifications to the above.

Satisfying this specification is equivalent to the process concerned refining $COUNT_0$ or $COUNT'_0$ as appropriate, where

$$\begin{aligned}
COUNT_0 &= up \rightarrow COUNT_1 \\
COUNT_{n+1} &= (STOP \sqcap up \rightarrow COUNT_{n+2}) \\
&\quad \Box\ down \rightarrow COUNT_n \\
COUNT'_0 &= up \rightarrow COUNT'_1 \\
&\quad \Box\ iszero \rightarrow COUNT'_0 \\
COUNT'_{n+1} &= (STOP \sqcap up \rightarrow COUNT'_{n+2}) \\
&\quad \Box\ down \rightarrow COUNT'_n
\end{aligned}$$

We therefore have a simple refinement check for both these predicates, but it is not finitary since the process on the left-hand side of the refinement is infinite state.[2] Note that there

[2] We note that, while the standard functionality of FDR is not able to deal with checks where the left-hand side is infinite state and the right-hand finite-state, it has long been known that this could be achieved by appropriate use of lazy normalisation, but it is suspected this would have a significant overhead in terms of running time.

are many finite-state processes that meet these specifications. For the time being we will concentrate on the first sort of counter (without *iszero*).

Approach 1: two-sided approximation

One solution to deciding whether a finite-state process is a counter using finite-state refinement checking is easily adapted from the approach set out in Chapter 5 of [6] for buffers. Define two series of processes:

- $COUNT^n$ is the most nondeterministic counter whose value never gets bigger than n. It behaves identically to *COUNT* when its value is less than n, but when n it cannot accept an *up*.
- $WCOUNT^n$ is the most nondeterministic *weak* counter of size n: it may accept an *up* that takes its value over n but if so it becomes equal to the most chaotic process (**div** $\sqcap$ *Chaos* works in all standard models).

These specifications are in the relationship

$$\ldots \sqsubseteq WCOUNT^n \sqsubseteq WCOUNT^{n+1} \ldots \sqsubseteq COUNT \sqsubseteq \ldots COUNT^{n+1} \sqsubseteq COUNT^n \ldots$$

It is then true that every finite state process which is a counter will refine one of the $COUNT^n$ and every finite-state process which is not a counter will fail to refine one of the $WCOUNT^n$.

Comparing a proposed counter against the specifications $WCOUNT^n$ and $COUNT^n$ for increasing values of n in turn therefore gives a decision procedure.

For obvious reasons, however, it would be nice to be able to resolve the issue in a small fixed number of checks.

Approach 2: constructive contexts

Recall that a CSP context $F(\cdot)$ is said to be *constructive*, or sometimes *guarded*, if for all processes P, Q and natural numbers n we have

$$P \downarrow n = Q \downarrow n \implies F(P) \downarrow (n+1) = F(Q) \downarrow (n+1)$$

where $P \downarrow n$ is the standard restriction to n steps of behaviour (see [6], for example). Recall also that every context built without hiding, where the process variable(s) only appear indirectly or directly guarded by some communication, is constructive.

Now suppose F is a constructive context which maps counters to counters, and that P is a process such that

$$F(P) \sqsubseteq P$$

If P were not itself a counter then there would be some shortest-length behaviour b which demonstrated this. Let the length of b be n in the sense that n is minimal such that $b \in Q \downarrow n$ if and only if $b \in Q$. n cannot be 0 since the only length 0 behaviour is the empty trace – which does not contradict being a counter – as both refusals and divergences on a trace of length k actually have length $k+1$. It follows straightforwardly that there is a process C such that C is a counter and

$$C \downarrow (n-1) = P \downarrow (n-1)$$

(To construct C simply build a process which behaves identically to P for the first $n-1$ steps and then behaves like $Count_r$ where the value of r is the excess of *up*s over *down*s in the

preceding trace.) But then $F(C)$ is a counter with the property that $F(C) \downarrow n = F(P) \downarrow n$. Since P has behaviour b and $P \sqsupseteq F(P)$ it follows that $F(P)$ has it too. Since $b \in R \downarrow n \Leftrightarrow b \in R$, it follows that $b \in F(C)$ which contradicts the fact that $F(C)$ is a counter.[3]

Hence P is a counter. Obviously this argument applies to any specification, not just counters. We will see another example later.

$$\begin{aligned} CT_A(P) &= up \to (P \underset{\{up\}}{\|} D) \\ D &= down \to RUN(\{up\}) \\ &\quad \Box\ (STOP \sqcap up \to D) \end{aligned}$$

is a good simple context for proving counters. For example $CT_A(P) \sqsubseteq P$ holds whenever P is one of the $COUNT^n$ bounded nondeterministic counters, or the deterministic processes with the same traces which can always be taken up to value n.

There are, however, counter processes which do not satisfy this refinement relation, meaning that CT_A gives a sound but not complete rule for proving counters. As an example consider

$$ExpC = up \to down \to COUNT^2$$

This has the trace $\langle up, down, up, up\rangle$ but $CT_A(ExpC)$ does not, since the first pair of events must have come from the context itself, and *ExpC* cannot initially perform a pair of *up*s.

Of course we can find a context that proves *ExpC* by giving it the ability to handle a second *up* itself, but it seems very unlikely to the author that any single constructive context can handle all counters. The grounds for this are that in general a counter can suddenly develop the ability to accept an arbitrary extra number of *up*s after any particular trace, but when P has done this trace the P(s) in $CT(P)$ has/have done a strictly shorter trace. He conjectures that this argument can be formalised into a proof of impossibility.

If we had adopted the second sort of counter specification (namely with *iszero*) then the context above can easily be adapted. Let

$$\begin{aligned} CT_B(P) &= up \to (P \underset{\{up,iszero\}}{\|} D') \\ &\quad \Box\ iszero \to CT_B(P) \\ D' &= down \to RUN(\{up,izsero\}) \\ &\quad \Box\ (STOP \sqcap up \to D') \end{aligned}$$

Then this has the same properties with respect to the new specification. Note that D' prevents P from communicating *iszero* until its own *down* has occurred.

Pseudo-constructive contexts

It is nevertheless possible to contract every nontrivial trace of a counter in a general and useful way. If t is a nonempty trace of a counter P then we can guarantee that it is also a trace of the process $P \,|||\, UD$, where $UD = up \to down \to STOP$, in such a way that only a trace t' of P that is strictly shorter than t is used in the interleaving. To see this note that the last *up* in the trace followed by a *down* if there is one following it might have come from *UD* however P has behaved.

[3] An alternative proof: as F is constructive it has a unique fixed point, which is the limit of the sequences $F^k(Q)$ for all processes Q in the standard metric space. If C is an arbitrary counter then so are all members of the sequence $F^k(C)$, and hence the limit. The limit of the sequence $F^k(P)$ is refined by P since $F(P) \sqsubseteq P$ and F is monotone. Since the two limits are the same it follows that P refines a counter, and so is a counter itself.

For the time being we will consider just the the traces of a possible counter process. There is of course a natural traces specification arising from the failures-divergences one:

$$t \downarrow up \geq t \downarrow down$$

and $t \in \{up, down\}^*$ for all traces t of the process we are considering. Call any process satisfying this a *trace counter*. However, while any counter satisfies this, failures-divergences counters actually satisfy stronger conditions on their sets of traces. For technical reasons that will soon be apparent, we will define a *strong trace counter* to be a process satisfying the above and which in addition, if t is a trace in which there are k less *down*s than *up*s, then $t^\frown\langle down\rangle^k$ is also a trace. (This is not a behavioural specification since it relates behaviours to each other, however every counter is a strong trace counter.)

Before turning to the failures aspects of counters we will try to capture the property of being a trace counter via a finitary refinement check.

If we could prove that

$$(P \;|||_C\; UD) \sqsubseteq_T P \qquad (\ddagger)$$

where $P \;|||_C\; UD$ consists of the empty trace together with all interleavings involving a non-empty trace of UD, then P is necessarily a trace-counter, as can easily be proved by induction on the length of trace. (We will see a similar induction in more detail below.)

Unfortunately $|||_C$ does not make sense as a CSP operator, and it is not implementable. In any implementation the first thing $P \;|||_C\; Q$ does has to come from Q, but it may need to have traces on which P performed the first action and Q didn't start till later!

We might term $P \;|||\; UD$ a *pseudo-constructive* context: it is not constructive but in some way has to play the role of one for us.

It is in fact possible to achieve the desired effect by applying suitable transformations to both sides of (‡). Specifically, we can identify the *up* that comes from UD by replacing UD on the left-hand side with

$$UD' = up' \rightarrow down \rightarrow STOP$$

Let $Counter'_L(P) = P \;|||\; UD'$.

On the right-hand side of the refinement we can use a fairly standard double renaming trick to rename the last *up* to *up*′:

$$\begin{aligned} Counter_R(P) &= P[\![\, up, up' / up, up \,]\!] \underset{\{up,up'\}}{\|} Reg \\ Reg &= up \rightarrow Reg \\ &\quad \Box\; up' \rightarrow STOP \end{aligned}$$

The renamed *up* becomes the last one because all subsequent ones are banned. Note that the last one may not be renamed, but if any it is the last one, and for all traces of P with an *up* there is a corresponding one of $Counter_R(P)$ where the last one is renamed.

Now consider the refinement $Counter'_L(P) \sqsubseteq_T Counter_R(P)$. Since P's traces are all in $Counter'_L(P)$ the refinement can only fail if one of the traces of $Counter_R(P)$ with its final *up* renamed is not in $Counter'_L(P)$. If P is a strong trace counter then this cannot happen thanks to the argument above. The "strong" here is necessary because of examples like

$$up \rightarrow up \rightarrow down \rightarrow down \rightarrow STOP$$

(a trace counter but not a strong one) can fail to satisfy it because of varying patterns of *down*s: here, $\langle up, up', down, down\rangle \notin traces(Counter'_L(P))$ because $\langle up, down\rangle \notin traces(P)$.

It even turns out that the refinement might hold if P is not a trace counter: it holds for example when $P = \mathit{down} \rightarrow \mathit{STOP}$. This latter example is one which needs to be dealt with.

To deal with this problem we replace $Counter'_L(P)$ by $Counter_L(P)$ where

$$\begin{aligned} Counter_L(P) &= Counter'_L(P) \underset{\Sigma}{\parallel} \mathit{UpFirst} \\ \mathit{UpFirst} &= ?x : \{\mathit{up}, \mathit{up}'\} \rightarrow \mathit{RUN}(\{\mathit{up}, \mathit{up}', \mathit{down}\}) \end{aligned}$$

Now, if the refinement

$$Counter_L(P) \sqsubseteq Counter_R(P)$$

holds it follows that the first event of P is *up*. All traces of P of length 0 and 1 are legitimate counter traces. Suppose the same is true of all traces of length k or less and that t has length $k+1$. Necessarily t has a last *up*, and the trace t' in which this is replaced by up' is a trace of $\mathit{Counter}_R(P)$. t' is therefore the interleaving of a trace t'' of P and either $\langle \mathit{up}' \rangle$ or $\langle \mathit{up}', \mathit{down} \rangle$. By induction $t'' \downarrow \mathit{down} \leq t'' \downarrow \mathit{up}$. It follows that the same is true of t.

It follows that $Counter_L(P) \sqsubseteq_T Counter_R(P)$ implies P is a trace-counter, and is implied by it being a strong trace counter.

If we move from the traces model to failures-divergences there is an obvious pathological case that passes the above refinement, namely **div**, the chaotic divergent process for which both $Counter_L$ and $Counter_R$ are strict. We can of course eliminate this possibility by a refinement check and it therefore makes sense to assume that P is known to be divergence-free.

In this case it is easier to create a refinement which establishes the correct failures specification if, noting that each counter is deadlock-free, we boost the divergence-freedom check to encompass deadlock-freedom (another one-state standard specification). Since the traces specification establishes that a trace counter with value 0 can only communicate *up* it follows that a deadlock-free trace counter cannot refuse *up* when 0. This is half of what we need, the other half being that it cannot refuse *down* when non-zero.

If we were simply to check $Counter_L(P) \sqsubseteq Counter_R(P)$ in the failures or failures-divergences model, this would make assertions about what P has to do in both *up*s and *down*s. The former would, as it turns out, be too strong. If, however, we define

$$Counter^{\mathcal{N}}_L(P) = Counter_L(P) \underset{\{up\}}{\parallel} \mathit{Chaos}_{\{\mathit{up}\}}$$

we get exactly what is required (in addition to the deadlock and livelock freedom check). The proof just extends the trace version above, once we note that if P can refuse *down* after a trace t with more *up*s than *down*s then $Counter_R(P)$ can do the same on t' where the last *up* has been replaced by up'. This is not permitted by the left-hand side since either the *UD* process offers *down* or the trace s which the left-hand P has performed is shorter and has more *up*s than *down*s.

It is possible to deal with the failures requirement on *up*s in a similar case-specific way by allowing *UD* to refuse up' if and only if P has performed an earlier *up*. *Reg* needs to be altered also. The details are left to the reader.

Since of course any trace counter which satisfies the failures requirements of a counter is also a strong trace counter, instead of the not-quite characterisation of trace buffers we achieved earlier, we now have a precise characterisation: a process P is a counter if and only if it is deadlock and divergence free, and satisfies

$$Counter^{\mathcal{N}}_L(P) \sqsubseteq Counter_R(P)$$

Thus we have reduced the decision procedure about whether a general process is a counter to two finitary refinement checks. These can, if desired, by combined into a single one using the representation of conjunctions introduced in [7].

Notice how the interplay between traces and refusals in this natural specification allowed us to achieve a tighter result than we could with traces alone.

It is interesting to note that if the $Counter_{L/R}$ refinement were to fail on FDR, then the way that the refinement works inductively coupled with FDR finding the shortest counterexample will generally mean that the example behaviour it produces will be a direct counterexample to P being a counter. (The shorter behaviour against which it is judged will, one would expect, behave correctly.) This may not be true, however, since the shortest route to an offending behaviour on the RHS may have a longer trace than others thanks to the way FDR handles τs. The way to solve this difficulty, should one encounter it, is to apply some τ-eliminating compression such as *diamond* [9] to $Counter_R(P)$ before the refinement check. If this is done it is guaranteed that the counterexample behaviour will directly contradict the counter specification.

The method developed in this section relied on us being able to find a systematic way to transform every trace t of a process satisfying a specification into a shorter one t' which can be lifted back to t by the constructive action of an operator (even though the operator itself might not be constructive) which preserves the specification. What we are doing is using a process as its own specification inductively, so that each finite state process has its own custom-made finite state specification.

Approach 4: watchdogs

In [2], we demonstrate how CSP behavioural specifications can be transformed to operate in parallel with the process being tested in such a way that if the specification fails then the parallel combination fails to refine a standard process which is independent of the problem. This form of specification is called a *watchdog* because it sits alongside the target process and 'woofs' when a mis-behaviour occurs. A typical watchdog for the counter specification is:

$$\begin{aligned} WatchC_0 &= (STOP \sqcap down \rightarrow woof \rightarrow STOP) \\ &\quad \Box\ up \rightarrow WatchC_1 \\ WatchC_{n+1} &= (STOP \sqcap up \rightarrow WatchC_{n+2}) \\ &\quad \Box\ down \rightarrow WatchC_n \end{aligned}$$

Note how these are very similar to the $COUNT_n$ processes.

If $WatchC_0 \underset{\{up,down\}}{\parallel} P$ is run for a counter P, notice that the values of the two processes will always coincide, and that the combination will never *woof* or deadlock. On the other hand hand any process with alphabet $\{up,down\}$ which fails the buffer specification will do one of these two things: an excess of *down*s will lead to a *woof*, and an illegal refusal of *up* (when 0) or *down* (when non-zero) will lead to deadlock.

One reason for our discussing watchdogs here is that, even though $WatchC_0$ is of course infinite state, if P is a finite-state counter then $WatchC_0 \underset{\{up,down\}}{\parallel} P$ is finite state (and can be run on FDR if care is taken to ensure the parallel combination is compiled at low level[4]: see

[4]For example, any $\checkmark$-free process P is compiled at low level if accessed as $low(P) = P;\ low(P)$. This is equivalent to P but contains a never-accessed recursion which forces low-level compilation. Since low-level compilation is slower and less space efficient than FDR's normal mode of running, it may be impractical to apply *low* to processes with large state spaces.

Appendix C of [6]). Notice however that if P is a finite-state process which is not a counter (such as $P = up \rightarrow P$, which is a trace counter) the parallel combination may be infinite state. Therefore this gives only a semi-decision procedure, and there are simple specifications where even this is not true.

Exactly the same applies to a counter with *iszero*. We quote the watchdog for this variant below. Note the way that, when one wants to test that a set of more than one elements are all available, each element has to be tested separately.

$$\begin{aligned} WatchC'_0 &= (STOP \sqcap down \rightarrow woof \rightarrow STOP) \\ &\quad \Box\ (up \rightarrow WatchC'_1 \\ &\quad\quad \sqcap\ iszero \rightarrow WatchC'_0) \\ \\ WatchC'_{n+1} &= (STOP \sqcap up \rightarrow WatchC_{n+2}) \\ &\quad \Box\ down \rightarrow WatchC_n \\ &\quad \Box\ (STOP \sqcap iszero \rightarrow woof \rightarrow STOP) \end{aligned}$$

Suppose C is a counter of this second sort. It is very easy to convert it into one if the first sort by $P \underset{\{iszero\}}{\parallel} STOP$. It is not nearly as easy to go the other way: if we have a context $C(P)$ which is operating on a counter of the first sort, the only clue that $C(P)$ has – by observing P – that P is in state 0 is that it refuses the event *down*. But there are no CSP operators which can introduce an extra event like *iszero* on the refusal of another: this would contradict monotonicity in trace sets. We conclude that the only way the context can know that it has to be able to say *iszero* is if it keeps a tally itself. The best way I can think of doing this is to put P in parallel with $Count'_0$: if P is a counter of the first sort then $P \underset{\{up,down\}}{\parallel} Count'_0$ is the corresponding counter of the second.

This method does, of course, have the same limitation as the watchdog, that it can only be guaranteed to produce a finite-state process if P is indeed a finite-state counter. (And is a failures/divergences counter, not just a trace counter.) Fortunately, of course, we have demonstrated in the previous section that it is possible to decide that question using finitary checks. So, as long as one is careful, infinite processes can be avoided for any finite-state P.

3 Processes with data

Suppose that instead of having events *up*, *down* and perhaps *iszero*, we now have the channels *in*, *out*, and perhaps the event *isempty*. The channels communicate values in some type T that we want to store inside our process. In this section we will study three related specifications:

- A *bag* is a process which holds data it has input on *in*, outputting each item exactly once on *out*. It never refuses input when empty and is always willing to output when nonempty. If it has the *isempty* then this only happens when the bag is empty (the items output are a permutation of those input) and cannot then be refused. A bag is not constrained as to which value to output if it has more than one.

- A *stack* is a bag which outputs on a strictly last-in first-out (LIFO) discipline. Of course the conventional names for the input and output channels of stacks are *push* and *pop*.

- A *buffer* is a bag which outputs on a strictly first-in first-out (FIFO) principle. Buffers have been extensively studied in CSP literature (generally without an *isempty* event).

Thus every stack and buffer is a bag, and the only process that is all three is *COPY* which can only hold a single item. Like counters, I will refer to first and second sort for ones that do and don't have *isempty*, and a trace bag (etc) will be one that just satisfies the trace constraints.

It is straightforward to adapt the counter failures divergences specification seen earlier into ones for each of these types of process. The buffer one can be found, for example, in chapter 5 of [6].

It is obvious that these three specifications are all close to that of a counter, and indeed the renaming

$$P[\![up, down, iszero / in.x, down.x, iszero \mid x \in T]\!] \qquad (\flat)$$

converts any bag into a counter. Of course my reason for using these specifications is that since we already know how to check if a finite-state process is a counter, then we can check the above transformation of any proposed bag is a counter as a large step on the way to verifying it. So all we have to do is concentrate on the data values that are passed around.

The most surprising discovery, however, is that under common conditions this transformation back to a counter is not necessary, because the introduction of data actually makes things easier as we shall see in the following section!

Data independence

A process is said to be data independent in the type T when it makes sense for any type T. It can input and output members of T, and store them. For different purposes we can allow further operations, such as implicit and explicit equality tests, constants, relational and functional symbols, and arrays indexed by T. See for example [3]. Note that data independence is usually considered as a syntactic restriction on programs rather than a semantic one, though [4] gives a semantic characterisation. While there is no guarantee that a process that is intended to be a bag, stack or buffer is data independent, there is every reason to believe that such programs will often be so.

The first mention of data independence in the literature, Pierre Wolper's paper [10], took a version of data independence at its purest without any of the add-ons such as equality tests. One of the main examples in that paper includes a way of testing whether a data independent process is a buffer (though the concept of a buffer is not exactly the same as ours as it does not place the same failures obligations on the process). Wolper's method is finitary, and can readily be adapted into a finitary way of deciding whether a finite-state data independent process[5] is a buffer in our sense, which is conceptually simpler than the way we already have of testing a counter.

The idea underlying this is that, since data independent P's behaviour does not depend on which values have been input into it (except in exactly which values are output), we can put special 'marker' values into the input stream and by ensuring that these come out at exactly the right points be sure that the buffer is correct.

If a divergence-free data independent process failed to be a buffer, it would be for one of the following reasons:

(a) It loses an input x, meaning either that y input after x comes out before x, or it fails to perform any further output at some time even though non-empty.

(b) It re-orders two inputs: this will manifest itself in the same way as the first version of (a).

(c) It duplicates a value x: x is output twice when it has only been input once.

[5] This means finite state when the type T is finite.

(d) It refuses to input when empty: this will mean it is possible either to refuse an input initially, or after it has just output the last value x that it has input.

(e) It refuses to output when it has input some value x but not output it yet. (Note that, thanks to divergence-freedom, this includes the second aspect of (a).

We can always input specific values once only amongst a background of constant values to represent x and y in all these clauses. The crucial thing about data independence is that doing so will not change the control flow of the process: it would do the same on any other stream of inputs. Suppose all values input are 0 except for one 1 followed an arbitrary time later by one 2: the inputs are prefixes of members of the regular language $0^*10^*20^*$ (as used also by Wolper). Then the above clauses reduce to:

(b) 2 is never output before 1 (for if any two values were treated as in (a) above then we could arrange that the 1 and 2 are input at the same points in the input stream as x and y respectively).

(c) At most one 1 appears in the outputs.

(d) The process can refuse any input either initially or when the last input and output were both 1.

(e) The process can refuse to output when it has input the 1 but not yet output it. (Note that the output it must give is either a 0 or

(a) Is covered by the combination of (b) and (e) as discussed above.

All of these things reduce to a single failures-divergences refinement check of the process $P \underset{\{|in|\}}{\parallel} Reg$, where

$$\begin{aligned} Reg &= in.0 \rightarrow Reg \\ &\quad \Box\ in.1 \rightarrow Reg' \\ Reg' &= in.0 \rightarrow Reg' \\ &\quad \Box\ in.2 \rightarrow Reg'' \\ Reg' &= in.0 \rightarrow Reg'' \end{aligned}$$

against the specification $DIBspec_0$, defined in Figure 1.

There, $DIBspec_0$ is the initial state (which must accept inputs); $DIBspec_1$ is the state where only 0's have been transacted so far; $DIBspec_2$ is the state where the last input was 1 and there have been no other inputs since; $DIBspec_3$ is the state where a 1 has been input followed by some 0's; $DIBspec_4$ is the state where the buffer is meant to be empty as the 1 from state $DIBspec_2$ (i.e., the most recent input) has just been output; $DIBspec_5$ is a more general state where the 1 has come out but the 2 not yet gone in, and we are not sure whether the buffer is empty or not; $DIBspec_6$ is a state where both 1 and 2 are in the buffer; $DIBspec_7$ is a state where 2 is in the buffer; $DIBspec_8$ represents the states where both 1 and 2 have been and gone.

At the expense of making *DIBspec* yet more elaborate (specifically, including a lot of extra events leading to **div** – for "don't care" – representing all the behaviours when the process does not follow the $0^*10^*20^*$ input pattern) this can be re-cast in the form $SPEC \sqsubseteq P$.

So data independence has removed the need for the self-referential refinement checking we encountered with counters in the last section. This is a rather remarkable fact. In essence

$$
\begin{array}{lcl}
DIBspec_0 & = & in.0 \to DIBspec_1 \\
 & & \Box\ in.1 \to DIBspec_2 \\
 & & \\
DIBspec_1 & = & STOP \sqcap \\
 & & (in.0 \to DIBspec_1 \\
 & & \Box\ in.1 \to DIBspec_2 \\
 & & \Box\ out.0 \to DIBspec_1) \\
 & & \\
DIBspec_2 & = & (STOP \sqcap \\
 & & (in.0 \to DIBspec_3 \\
 & & \Box\ in.2 \to DIBspec_6)) \\
 & & \Box \\
 & & (out.0 \to DIBspec_2 \\
 & & \sqcap\ out.1 \to DIBspec_4) \\
 & & \\
DIBspec_3 & = & (STOP \sqcap \\
 & & (in.0 \to DIBspec_3 \\
 & & \Box\ in.2 \to DIBspec_6)) \\
 & & \Box \\
 & & (out.0 \to DIBspec_3 \\
 & & \sqcap\ out.1 \to DIBspec_5) \\
 & & \\
DIBspec_4 & = & (in.0 \to DIBspec_5 \ \Box\ in.2 \to DIBspec_7) \\
 & & \\
DIBspec_5 & = & STOP \\
 & & \sqcap\ (in.0 \to DIBspec_5 \ \Box\ in.2 \to DIBspec_7) \\
 & & \sqcap\ out.0 \to DIBspec_5 \\
 & & \\
DIBspec_6 & = & (out.0 \to DIBspec_6 \sqcap out.1 \to DIBspec_7) \\
 & & \Box\ (STOP \sqcap in.0 \to DIBspec_6) \\
 & & \\
DIBspec_7 & = & (out.0 \to DIBspec_7 \sqcap out.2 \to DIBspec_8) \\
 & & \Box\ (STOP \sqcap in.0 \to DIBspec_7) \\
 & & \\
DISspec_8 & = & Chaos_{\{in.0,out.0\}}
\end{array}
$$

Figure 1: Specification for data independent buffer check

this is because we are able to get more of a handle on the number of pending outputs thanks to tagging them.

Those experienced in data independence will recognise that under the strong conditions we have assumed on our implementation, it is normally sufficient to check with T of size 2, whereas the above check uses one of size 3. The only property for which we have relied on 2 is establishing the absence of re-ordering. It is in fact possible to detect errors of this and the other forms by using the language $0^*(1+11)0^*$, thereby cutting the type to $\{0,1\}$.

Subject to the remark in the next paragraph, it is easy to modify *DIBspec* to test $P \underset{\{|in|\}}{\|} Reg$ (with P data independent) for being a stack. Bags are easy because all one needs to do is to put a single 1 in amongst a stream of zeros and ensure (a) that 1 comes out no more than once and (b) that the bag cannot refuse to output while it contains the 1. Notice that 1 only coming out once implies (since all the output values were previously input somewhere) that the total number of outputs never exceeds that of inputs.

Neither the bag nor stack specifications described here address the question of forcing the processes to input when empty. This is because it is not possible for the specification to detect when the process is empty in a finite-state way, so it specially forces input. There is, however, a simple solution to this (also possible for buffers), namely modifying each specification state so it does not accept any deadlock.

Bags and stacks without data independence

In this section we see what can be done without the assumption of data independence. The failures-divergences bag and stack specifications can be established with relatively minor modifications to the techniques used for counters. The first thing to remark is that there will be no problems with the refusal parts of the failure specification of a bag: for the *out.x* events the requirements are the same as for *down* events in counters. And for the requirement that a bag will input anything when empty all one has to do (in conjunction with proving it is trace-correct) is to show it refines:

$$IODF = (in?x \rightarrow IODF) \sqcap (\sqcap_{x \in T} out!x \rightarrow IODF)$$

In other words it can always either output, or accept all inputs.

Since a trace stack or trace buffer is respectively a stack or buffer if and only if it is a bag, we can therefore restrict ourselves to deciding whether a process is a bag, and whether it is a trace stack or buffer.

A little thought reveals that a bag is precisely a process P satisfying the strengthened deadlock freedom check above (also proving divergence freedom, of course), for which (a) the renaming given as $(\flat)$ above satisfies the counter check

$$Counter_L^{\mathcal{N}}(P^\flat) \sqsubseteq Counter_R(P^\flat) \qquad (\sharp)$$

and (b) such that for each member x of T the process

$$(C_x(P) = P \setminus \{in.y, out.y \mid y \neq x\})[\![{}^{up,down}/_{in.x,\, out.x}]\!]$$

is a trace counter. Since any process satisfying this also has all the $C_x(P)$ strong trace counters, (b) is equivalent to $\sqcap\{C_x(P) \mid x \in T\}$ satisfying the trace refinement version of $(\sharp)$.

The above check is less efficient than I would have liked since in essence it involves examining P's state space $n+1$ times, where n is the size of T. To see why something like this seems to be necessary, and also to help our understanding of what can and cannot be

done with pseudo-constructive contexts, I will now go over an earlier attempt at specifying bags that did not quite work.

It is reasonably obvious (and readily proved) that if B is a bag then so is $B \,|||\, IO$, where $IO = in?x \to out!x \to STOP$. Clearly every nonempty trace of a bag starts with an event of the form $in.x$. It is tempting to believe that nonempty traces of B are also, analogously to counters, traces of $B \,|||\, IO$ in which the last input of s is performed by IO (so that the interleaved copy of B performs a strictly shorter trace).

To see that this is not in general true let $B^?_K$ be a bag that uses a strict stack regime to output its pending values until its contents grow to size K for the first time, after which it for ever after outputs as though it were a buffer (including the outputs pending at the time). If $K = 3$ it will have the trace

$$\langle in.1, in.2, in.3, out.1, out.2, out.3\rangle$$

but $\langle in.1, in.2, out.1, out.2\rangle$ is not a trace, which it would have to be for the desired property to hold.

Let's define a *stable* bag to be one where, if $t^\frown\langle in.y, out.x_1, \ldots, out.x_n\rangle$ is a trace, then so is $t^\frown\langle out.y_1, \ldots, out.y_m\rangle$ where $m \in \{n-1, n\}$ and the y_i are a subsequence of the x_i obtained by deleting (at most) $out.x$. (In the case where several x_i are equal to x all that is necessary is that there is one subsequence with this property.) In other words the order our process outputs values it had in it at the point where it performs the input $in.x$ was also possible before that input. Note that $B^?_3$ is not stable.

The following construction straightforwardly implies that any nonempty trace of B is one of $B \,|||\, IO$ in which the last input was performed by IO.

So if we define

$$\begin{aligned}
IO' &= in'?x \to out!x \to STOP \\
InFirst &= in?x \to RUN_{\{|in,in',out|\}} \\
&\quad \Box\ in'?x \to RUN_{\{|in,in',out|\}} \\
Bag_L(P) &= (P \,|||\, IO') \underset{\{in,in',out\}}{\parallel} InFirst \\
Reg &= in?x \to Reg \\
&\quad \Box\ in'?x \to STOP \\
Bag_R(P) &= P[\![\, in, in' / in, in \,]\!] \underset{\{|in,in'|\}}{\parallel} Reg
\end{aligned}$$

Then $Bag_L(P) \sqsubseteq_T Bag_R(P)$ if P is a stable strong traces bag and implies it is a stable traces bag. The proof of this is essentially the same as for counters.

The need for a condition such as stability in arises here because pseudo-constructive specifications, by their nature, have to reduce every nonempty trace of a process to a shorter one, so there cannot be any new sorts of behaviour suddenly appearing after a long trace. Note that, like our strong trace conditions, the stable bag condition is not a behavioural predicate.

It is easy to adapt the stable bag construction to characterise stacks. (Note that a stack is a stable bag.) What we know there is that as soon as the last input of a trace has gone in, the next input if any is tied to this input. So, whereas in $Bag_L(P)$ the output from IO' can occur any time after the input, we will now have to require that it is the next output. The easiest way to achieve this is to prime the output from IO, replacing it by

$$SIO = (in'?x \to out'x \to RUN_{\{|out|\}}) \,\Box\, (out?x \to SIO)$$

and defining

$$Stack_L(P) = ((P \underset{\{|out|\}}{\parallel} SIO)[\![out / out']\!]) \underset{\{in,in',out\}}{\parallel} InFirst$$

We can then set $Stack_R(P) = Bag_R(P)$, and have the expected result.

Buffers

I have been unable to find a similar and satisfactory construction to the above for buffers. The difficulty comes from the following:

- In looking for a pseudo-constructive context to characterise any of the sorts of specifications we have looked at, it is necessary to have the constructive action (i.e. the insertion of extra event(s)) towards the end of the trace. This is because the insertion towards the beginning of the trace (as was done in the section on constructive constructs above and which works equally well for buffers: see below) might change the LHS behaviour in a way that the pre- and post-insertion behaviours do not line up well enough for comparison.
- Given that we are inserting an input and perhaps output towards the end of a buffer trace, there seems (unlike with stacks) to be no straightforward way of placing the extra output in the right place.

So the best I have been able to do on previous models was to give a sound and incomplete rule, namely that P is a buffer if it satisfies

$$PB(P) \sqsubseteq P$$

where

$$\begin{aligned} PB(P) &= in?x \to (P \underset{\{in,out\}}{\parallel} Z(x))[\![out / out']\!] \\ Z(x) &= (STOP \sqcap in?y \to Z(x)) \\ &\quad \Box\ out'!x \to RUN_{\{in,out\}} \end{aligned}$$

This is not a complete rule because it cannot handle buffers which expand on the first output, for example

$$ExpB = in?x \to out!x \to B_2$$

where B_2 is a two-place buffer. Clearly this is essentially the same problem we found with counters when using the context CT_A.

Note the similarity of the PB-based rule to the rather more complex buffer laws $BL5$ and $BL5'$ from chapter 5 of [6] (originally proved in [8]) which apply to piped combinations rather than single processes.

The only way I have found to solve the problem of placing the last (rather than first) input into the correct place in the input stream borrows from the idea described earlier for turning a counter of the first sort into one of the second.

The first thing one has to do (to ensure that this exercise does not fail to terminate) is to check either that when renamed as in $(\flat)$ it is a counter. The left-hand side process is then constructed by running P in parallel with IO and the most nondeterministic buffer $BUFF'_{\langle\rangle}$ of the second sort. All the inputs of this process prior to IO's input are fed into $BUFF'$, but not that input or any subsequent one. Outputs of P prior to IO's output are synchronised with

$BUFF'$. IO's output is permitted once $BUFF'$ has become empty (which we can see thanks to the *isempty* event, which is synchronised with this output). Once this extra output has appeared then P is allowed to output freely again.

This generates a finite-state process which (with the extra input suitably renamed as usual) can be used to check the same right-hand side process we used for bags and stacks.

I feel, however, that this approach is clumsy and inelegant, and that it is less than appealing because of the way it uses an infinite state process in its definition. Fortunately, however, a different approach does give an elegant formulation.

Stacks and buffers share an interesting property that general bags do not have, namely, after any trace where an output is available, that output is completely determined by the trace. We can define a bag with this property to be *output deterministic*, noting that this certainly does not imply that the process itself is deterministic.

Lazić [3] developed a technique, reworked in [7], for establishing the determinism of a process via a finitary refinement check. Let P' be a copy of P renamed so that each event a in its alphabet A becomes $a' \in A'$, where this renaming is a bijection and $A \cap A' = \{\}$, and let $Test = ?a : A \rightarrow a' \rightarrow Test$. Then P is deterministic if and only if

$$(P \;|||\; P') \underset{A \cup A'}{\parallel} Test$$

can never deadlock after an odd-length trace. This idea can be modified to produce a test for output determinism: let

$$\begin{array}{lcl} TestOD & = & out?x \rightarrow out'?y \rightarrow (TestOD \lhd x = y \rhd error \rightarrow STOP) \\ & & \Box\ in?x \rightarrow TestOD \end{array}$$

and consider

$$(P \underset{\{|in|\}}{\parallel} P[\![\, out' / out \,]\!]) \underset{\{|in,out,out'|\}}{\parallel} TestOD$$

This makes sure that the two copies of P have always done the same sequence of inputs and outputs, but if they do give different outputs at any time the event *error* appears. Evidently the combination refines $Chaos_{\{|in,out,out'|\}}$ precisely when P is output deterministic. So we can check if a process is an output deterministic bag via a few finitary refinement checks.

Recall that the CSP operator $P \gg Q$ connects the outputs of P to the inputs of Q and hides these internal communications. If P is a buffer then so is $CP = COPY \gg P$, where

$$COPY = in?x \rightarrow out!x \rightarrow COPY$$

is the simple one-place buffer (this is a consequence of $BL1$ in chapter 5 of [6]). CP is therefore output deterministic.

On the other hand, if P (known to be a bag) is not a buffer, then the only way this can happen is if it has a trace $s^\frown\langle out.x\rangle$ where x is the wrong value to be output (i.e. is not the next pending one). We can assume that this is a shortest such trace. Necessarily, after s (which is a buffer trace), there are at least two values pending since if there were only one the bag P would have to output the buffer-correct one. Now s is also a trace of CP, and this can happen in two ways:

- *Either COPY* transmits all its inputs directly to P so that in the final state P has performed the trace s;
- *or COPY* has retained the very last input $in.y$ in s and not yet transmitted it to P. P can perform the trace s' which is s with its last input deleted since all outputs after this input are pending before that input occurs, and there is no shorter counterexample than $s\langle out.x\rangle$ to buffer behaviour. Note that, thanks to what we observed above, P is then (i.e., after s') in a state where it has a pending output.

In the first of these states CP can evidently output $out.x$ because P then can. In the second of the states it can, since P has performed a trace strictly shorter than s, output the correct $out.z$ based on FIFO in s. It follows that CP is not output deterministic.

An elegant decision procedure for P being a buffer is then to test if it is a stable bag (since all buffers are stable) and then to check that CP is output deterministic.

It is similarly possible to find out if a bag P is a stack by testing if a different context around P is output deterministic. This is left as an exercise to the reader.

4 The problem of generalisation

It is interesting to see what lessons we can learn from these examples and attempt to get some general results. For simplicity I will mainly consider trace specifications.

Any simple trace refinement is equivalent to the question of whether a given regular language (the traces of a finite-state process) is a subset of some set S of traces. Clearly the case in which the specification is also a finite-state process reduces to the question of whether one regular language is contained in another. However none of the cases we have been studying in this paper have S regular.

The traces of counters and stacks are prefix-closed context-free grammars, whereas the languages of buffers and bags are not. It seems very unlikely that prefix-closed subsets of any context-free language can be decided by a pseudo-constructive trace refinement (which is all we ought to be using to decide languages of traces, of course). We already had difficulties with, for example, weak counters (arbitrary prefix-closed subsets of the set of all counter traces), and consider for example the context-free language

$$a^*b^* \cup \{a^n b^n c \mid n \in \mathbb{N}\}$$

It seems very unlikely to me that a process's traces being contained within this is capturable by a finitary refinement using a pseudo-constructive context, bearing in mind that a trace of the form $a^n b^n c$ (which may, for arbitrarily large n, be the only such trace in our process) can be reduced meaningfully to another one of the same process. It can, however, be decided by a watchdog using the trick seen earlier which allows the inclusion of an infinite-state watchdog into a finite-state system.

We have already seen that the failures aspects of specifications can have significant effects: for example we were unable to characterise trace counters but were able to characterise failures-divergences one because of the interplay between refusals and traces. It almost seemed that the "naturalness" of our failures specifications, particularly the way they force a process to be reducible to zero/empty, played a big role in making them accessible.

There are obviously many different CSP contexts we could use which could play a pseudo-constructive role. In order to get any sorts of general characterisation of the specifications one could create using them we would have to understand the sorts of grammatical constructors they represent. This is something which needs to be investigated, but I have no idea if there is any sort of clean result here.

5 Conclusions and future work

We have seen how to characterise a number of related infinitary properties and seen a number of tricks for expressing them. While all the properties we aimed to characterise were behavioural specifications, we have found ourselves forced to use non behavioural approximations to them such as strong trace counters and stable bags.

While it is not too hard to see *why* data independence gave us extra power, the fact still seems rather wonderful and is certainly worthy of deeper investigation.

This work has brought extra insights into the nature of things like buffers and bags and potentially useful concepts such as stable bags and output determinism. I hope that by using these and the new specification techniques being developed I can shed more light on the important but difficult topic of *buffer tolerance* (see chapter 5 of [6] for example). It seems likely that the ideas like output determinism will be useful. Certainly it is a non-behavioural property which is of huge practical importance.

The ultimate goal is some sort of logical or mathematical characterisation of what properties are expressible as finitary refinement checks. The expressive power of CSP, coupled with the self-referential nature of the checks themselves (illustrated by our pseudo-constructive technique) make this a considerable challenge.

Acknowledgements

I would like to thank Gavin Lowe for his comments on an earlier draft of this paper.

References

[1] Formal Systems (Europe) Ltd., *Failures-Divergence Refinement*, User Manual, obtainable from `www.fsel.com/documentation/fdr2/html/`

[2] M.H. Goldsmith, N.Moffat, A.W. Roscoe, T. Whitworth and I. Zakiuddin, *Watchdog transformations for property-oriented model checking*, Proceedings of FME 2003.

[3] R.S. Lazić, *A semantic study of data-independence with applications to the mechanical verification of concurrent systems*, Oxford University D.Phil thesis, 1998.

[4] R.S. Lazić and D. Nowak, *A unifying approach to data-independence*, In Proceedings of the 11th International Conference on Concurrency Theory (CONCUR 2000), Lecture Notes in Computer Science. Springer-Verlag, August 2000.

[5] J.N. Reed, J. Sinclair and A.W. Roscoe, *Responsiveness of Interoperating Components*, To appear in FAC.

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

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

[8] A.W. Roscoe, *A mathematical theory of communicating processes* Oxford University D.Phil thesis, 1982.

[9] A.W. Roscoe, P.H.B. Gardiner, M.H. Goldsmith, J.R. Hulance, D.M. Jackson and J.B. Scattergood, *Hierarchical compression for model-checking CSP* or *how to check* 10^{20} *dining philosophers for deadlock*, Proceedings of the 1^{st} TACAS, BRICS Notes Series NS-95-2, Department of Computer Science, University of Aarhus, 1995. (Also Springer LNCS 1019.)

[10] P. Wolper. Expressing interesting properties of programs in propositional temporal logic. In *Proceedings of the 13th PoPL* , pages 184–193, 1986.