The expressiveness of CSP extended by priority (draft)

A.W. Roscoe

Oxford University Department of Computer Science

October 17, 2014

Abstract

In previous work [27, 26] the author defined a notion of *CSP-like* operational semantics whose main restrictions were the automatic promotion of most τ actions, no cloning of running processes, and no negative premises in operational semantic rules. He showed that every operator with such an operational semantics can be translated into CSP and therefore has a semantics in every model of CSP. In this paper we demonstrate that a similar result holds for CSP extended by the priority operator described in Chapter 20 of [26], with the restriction on negative premises almost completely removed. However, since priority is not compositional in most CSP models, the range of such models that support operators with negative premises is limited.

1 Introduction

While other languages for concurrent systems are often defined in terms of their operational semantics, the CSP approach [13, 22, 26] has always been to regard behavioural models such as *traces* $\mathcal{T}$ and *failures-divergences* $\mathcal{N}$ as at least equally important means of expression. Thus any operator must make sense over these *behavioural* models in which details of individual linear runs of the processes are recorded by an observer who cannot, of course, see the internal action τ.

Nevertheless CSP has a well-established operational semantics first described in SOS in [5, 7], and congruence with that is perhaps the main criterion for the acceptability of any new semantic model.

Operational semantic definitions of languages have the advantage that they are direct, understandable, and of themselves carry no particular obligation to prove congruence results such as those alluded to above. On the other hand definitions in abstract models, intended to capture the extensional meaning of a program in some sense, have the advantage of "cleanliness" and allow us to reason about programs in the more abstract setting. The most immediate advantages of CSP models in this respect is that they bring a theory of refinement which in turn gives refinement checking (with low complexity at the implementation end, as in FDR[1] as a natural vehicle for specification and verification.)

In earlier work, the author created a class of *CSP-like* operational semantic definitions that automatically map congruently onto definitions over the whole class of CSP models, thereby giving both sets of advantages as well as freeing the language designer from the need to prove congruence theorems.

In addition to a number of other restrictions on the full generality of Structured Operational Semantic (SOS) definitions, CSP-like ones are not permitted any negative premises: thus there can be no rule in which a given action can fire only if one its arguments *can not* perform some (either one or more) action(s).

There have been a number of different proposals for adding priority to CSP. A relatively straightforward one, in part because it does not involve building special semantic models or types of LTSs, was proposed in [26]. $\mathbf{Pri}_{\leq}(P)$, for a partial order on the events that processes perform, permits P only to perform event x only when no higher priority event is possible for P.[2] With a few restrictions on how the invisible event τ and termination signal *tick* fit into $\leq$, this adds very usefully to the expressive power of CSP, for example by permitting the accurate description of real-time systems.

$\mathbf{Pri}_{\leq}(P)$ is certainly not CSP-like in the sense alluded to above, since it requires negative premises. Indeed it does not have a semantics in most CSP models, requiring the richest sort of model which potentially record an acceptance/ready set before every visible event: essentially all behaviours recordable in finite linear fashion by an external observer. In [26], this class[3] of models is based on the version $\mathcal{FL}$ (standing for finite linear) that records only finite behaviour. These models are sometimes termed "ready traces".

[1] FDR[12, 10] is a model checker/refinement checker for CSP.

[2] For reasons that will be explained later, it is better to restrict $\leq$ to be an order without any infinite ascending chains. In other words its reversal should be well-founded.

[3] This means models where the finite behaviours recorded of processes are those of the model $\mathcal{FL}$: there is some choice over what infinite behaviour if any is recorded: see [26]. We will give more details of *FL* later in this paper.

This naturally begs the question of whether we can capture a notion of *Pri-CSP-like* operational semantics which includes this operator and all of which can be expressed in terms of CSP with it added. Establishing such a notion is the job of the present paper.

In the next section, we remind ourselves about the CSP language and its operational semantics. We then recall the definition of CSP-like operational semantics and the outline of the expressiveness result involving it. Finally we recall the definitions of $\mathbf{Pri}_{\leq}(\cdot)$ in terms of operational semantics and over $\mathcal{FL}$.

In Section 3 we generalise the definition of *CSP-like* to what we expect to be able to express with the addition of $\mathbf{Pri}_{\leq}(\cdot)$. The main result of this paper then follows, in which we show that any operator (or class of operators) with such Pri-CSP-like operational semantics, can be simulated precisely in CSP thus augmented. As with the previous result, the degree of precision obtained by this simulation depends on whether or not the language involves the CSP concept of termination, represented by the special event $\checkmark$.

Whereas CSP-like languages have semantics over all CSP models, Pri-CSP-like ones can only, in general, be guaranteed to have semantics over the $\mathcal{FL}$ class of models of programs to be verified on FDR provided it is extended to cope with such models.

In Section 5 we discover that our generalised notion of combinator operational semantics does not have a nice compositional property that the original version does. We also explore extensions to the sorts of premises we allow on processes, and discuss an alternative *generalised* form of Pri-CSP-likeness which not only allows one to test negative premises, but also check that a stable process can perform any of a set of events.

2 Background

2.1 The operational semantics of CSP

The standard operational semantics of CSP came along after the most common denotational semantics, in [5, 7]. Technically speaking, this is primarily a paper about these operational semantics as extended by more recent developments. In common with other similar languages, CSP's operational semantics generate labelled transition systems (LTSs). For CSP, the action labels come from $\Sigma \cup \{\tau, \checkmark\}$, where Σ is the *alphabet*, the set of actions that are visible to and controllable by the external observer, τ is an invisible and uncontrollable event such that whenever it is enabled and another event does not happen quickly, it will. $\checkmark$ is a signal representing the termination

of a process. It is observable, but it is generally better to think of it as uncontrollable in the same sense as τ.

The whole class of closed (i.e., no free identifiers) CSP terms over a given alphabet can be thought of as a large LTS. While tools like FDR can only handle finite CSP terms over finite alphabets, from a theoretical point of view we are happy to envisage infinite alphabets and infinitary terms (though with well-founded syntax) created by infinite mutual recursions, and infinite nondeterministic choice.

In this paper we permit, and indeed require in certain circumstances, infinite alphabets, though aspects of it are easier for finite ones.

Given the process P, αP means its own set of Σ actions, which is usually just the ones it uses, and when our process P appears in an alphabetised parallel $P \ {}_X\|_Y \ Q$ it is invariably X. Hoare [13] makes such alphabets an important part of his semantics of CSP. We, following [22, 26], use it as a less formal notation.

The CSP language we use in this paper has two operators not seen until relatively recently. These are the *throw* operator $P \, \Theta_A \, Q$ [23] where if P communicates $a \in A$ this hands control to Q, and the priority operator that is in some sense the main topic of this paper. Both of these operators add significantly to the expressive power of CSP. We are studying that of priority in this paper; the original motivation for Θ_A can be found in [23].

Below we introduce the language, including Θ_A, via SOS operational semantics [19].

In SOS style we need rules to infer every action that each process can perform when applied to its appropriate number of arguments. The conditions that enable actions are always, so far as we are concerned, of three sorts:

- *Positive:* Some other process can perform a specific action. This other process is determined from the syntax of the process P whose transitions we are calculating. In our setting these other processes are, except in the case of recursion, arguments of the operator whose semantics we are defining.

- *Negative:* The same except the other process cannot perform a given action.

- *Side conditions* on the actions, alphabets etc that appear free in the rule. In this paper we regard each instance of a rule for different such free objects as a different rule, formally speaking.

For us, therefore, a rule comes with set of action/alphabet etc parameters, and appropriate positive and negative *premises*. We can think of a rule with free parameters other than processes as a rule *schema*.

CSP is a language that consists of a few constant processes, a number of operators which can be applied to one or more argument processes to create another one, and recursive constructions. The operational semantics of constants simply describe their actions directly. Thus $STOP$, which has no actions, has no operational rules, and $SKIP$, the process which simply terminates immediately, has the single rule

$$\frac{}{SKIP \xrightarrow{\checkmark} \Omega}$$

A process does nothing after terminating. Ω is a "process" we see precisely after ✓ actions, essentially representing something we do not bother to look at because the process is finished. In some ways it would be better to use the notation $P \xrightarrow{\checkmark}$ rather than $P \xrightarrow{\checkmark} Q$.

Other important constant processes are RUN_A, which performs any sequence of events from $A \subseteq \Sigma$ and never refuses one, $Chaos_A$ which is the most nondeterministic non-divergent process on the events A which can always pick whatever subset it chooses of A to offer, and **div** which simply *diverges* (i.e. performs an infinite unbroken series of τs).

There are two choices of how to handle recursive terms operationally, which are summarised by the rules

$$\frac{}{\mu\, p.P \xrightarrow{\tau} P[\mu\, p.P/p]}(A) \qquad \frac{P[\mu\, p.P/p] \xrightarrow{x} Q}{\mu\, p.P \xrightarrow{x} Q}(B)$$

where $\mu\, p.P$ is the same as the recursive value calculated by the equation $p = P$, for p a process identifier and P a process term in which p may be free. Rule (A) introduces a τ every time a recursion is unwound, and Rule (B) does not. Thanks to the CSP principle that the process $\tau.P$ (in CCS notation: one that performs a τ before becoming P) is equivalent in all but operational semantics to P there is no observable difference between the results of these two rules, provided (B) is well defined. For a clean mathematical analysis of operational semantics, (A) is better as the τ guards eliminate problems caused by under-defined recursions (of which the simplest example is $\mu\, p.p$), which become more severe in the presence of the negative premises we will be considering in this paper. We will give an example to illustrate this once we have formally defined $\mathbf{Pri}_{\leq}(\cdot)$.

On the other hand, without such an undefined recursion (one where the first-step action s of a recursive body $P[Q/p]$ are not independent of those

of Q, or where the derivation of actions in an infinite mutual recursion is not well founded, as with the recursion $P_i = P_{i+1} \Box a \to STOP$), such problems do not arise and (B) gives a more efficient LTS representing any term. In this paper, for simplicity (not only with negative premises) we generally assume approach (A) in any case where it cannot be determined simply that every recursive call is guarded by at least one action (which can be τ), and the more efficient (B) otherwise. Parts of our later constructions rely on the use of (B) for the obviously guarded processes used in simulations: with (A) our simulations would not be as tight as claimed in the theorems.

The bulk of the operational semantics of CSP is concerned with operators such as $a \to P$ (prefix) and $P \underset{X}{\parallel} Q$ (parallel). The following clauses are taken from [22, 26]. In each case the transitions on the bottom are enabled just when all of the preconditions on the top are true, and the transitions of any term are the minimal set consistent with the rules stated.

2.2 The transition rules of CSP operators

The main way communications are *introduced* into the operational semantics is via the prefixing operation $e \to P$. In general, e may be a complex object, perhaps involving much computation to work out what it represents. The prefix e may represent a range of possible communications and bind one or more identifiers in P, as in the examples

$$?x : A \to P \qquad c?x?y \to P \qquad c?x!e \to P$$

We thus assume the existence of functions *comms* and *subs*.

- $comms(e)$ is the set of communications described by e. For example, $d.3$ represents $\{d.3\}$ and $c?x{:}A?y$ represents $\{c.a.b \mid a.b \in type(c), a \in A\}$.
- For $a \in comms(e)$, $subs(a, e, P)$ is the result of substituting the appropriate part of a for each identifier in P bound by e. This equals P if there are no identifiers bound (as when e is $d.3$). For example,

$$subs(c.1.2, c?x?y, d!x \to P(x, y)) \;=\; d!1 \to P(1, 2)$$

The transition rule for prefix is then easy to state:

$$\frac{}{e \to P \xrightarrow{a} subs(a, e, P)}(a \in comms(e))$$

It says what we might expect: that the initial events of $e \to P$ are $comms(e)$ and that the process then moves into the state where the effects of any inputs in the communication have been accounted for.

CSP nondeterministic choice is an operator that can take a τ action to either of its arguments: it alone gets to choose which to act like.

$$\frac{}{P \sqcap Q \xrightarrow{\tau} P} \qquad \frac{}{P \sqcap Q \xrightarrow{\tau} Q}$$

This easily translates to a generalized notion of choice $\sqcap S$ over a non-empty set S of processes:

$$\frac{}{\sqcap S \xrightarrow{\tau} P} \quad (P \in S)$$

All the other operators have rules that allow us to deduce what actions a process of the given form has from the actions of the sub-processes. Imagine that the operators have some of their arguments 'active' and some 'inactive'. The former are the ones whose actions are immediately relevant, the latter the ones which are not needed to deduce the first actions of the combination. (All the arguments of the operators seen above are initially inactive.) This idea comes across most clearly in the construct $P;\ Q$ (whose operational semantics can be found below), where the first argument is active, but the second is not as its actions do not become enabled until after the first has terminated.

Both the arguments of external choice ($\Box$) are active, since a visible action of either must be allowed. Once an argument is made active, it must be allowed to perform any τ or $\checkmark$ action it is capable of, since the argument's environment (in this case the operator) is, by assumption, incapable of stopping them. There is, however, a difference between these two cases since a τ action is invisible to the operator, which means that there are always rules like the following

$$\frac{P \xrightarrow{\tau} P'}{P \Box Q \xrightarrow{\tau} P' \Box Q} \qquad \frac{Q \xrightarrow{\tau} Q'}{P \Box Q \xrightarrow{\tau} P \Box Q'}$$

which simply allow the τ to happen without otherwise affecting the process state. (In some cases these rules are implied by more general ones.) These rules simply *promote* the τ action of the arguments to τ actions of the whole process. On the other hand, the $\checkmark$ event is visible, so (as with other visible actions) the operator can take notice and, for example, resolve a choice. With $\Box$, there is no difference in how $\checkmark$ and other visible events are handled:

$$\frac{P \xrightarrow{a} P'}{P \Box Q \xrightarrow{a} P'}(a \neq \tau) \qquad \frac{Q \xrightarrow{a} Q'}{P \Box Q \xrightarrow{a} Q'}(a \neq \tau)$$

There is an additional and theoretically useful choice operator $P \triangleright Q$ which can be characterised as an "untimed timeout" or "sliding choice". It initially offers choices provided by P, and if a visible one occurs this resolves the choice. But there is a τ action which will resolve the choice in Q's favour if P does not quickly do this. P is initially active:

$$\frac{P \xrightarrow{\tau} P'}{P \triangleright Q \xrightarrow{\tau} P' \triangleright Q}$$

Any visible action from P decides the choice in its favour

$$\frac{P \xrightarrow{a} P'}{P \triangleright Q \xrightarrow{a} P'} \quad (a \neq \tau)$$

The said τ can resolve the choice in Q's favour:

$$\frac{}{P \triangleright Q \xrightarrow{\tau} Q}$$

Of course, the place where $\checkmark$ is most important is in the sequential composition operator ; . Here, the first operand is necessarily active, while the second is not. In $P;\ Q$, P is allowed to perform any action at all, and unless that action is $\checkmark$ it has no effect on the overall configuration.

$$\frac{P \xrightarrow{x} P'}{P;\ Q \xrightarrow{x} P';\ Q} \quad (x \neq \checkmark)$$

If P does perform $\checkmark$, indicating it is terminating, this simply starts up Q, with the action itself being hidden from the outside – becoming τ.

$$\frac{P \xrightarrow{\checkmark} P'}{P;\ Q \xrightarrow{\tau} Q}$$

It is semantically important that the second argument of ; and the process argument of $e \rightarrow P$ are inactive, for if they were not, they would be allowed to perform any τ actions so that if they could diverge, so could the overall process. And the process $STOP$; **div** (**div** being the divergent process described earlier) could never get into a stable state even though it is supposed to be equivalent to $STOP$. This shows that any argument which is active is always one in which the operator is divergence-strict (i.e., maps immediately divergent processes to immediately divergent processes).

The rules for hiding and renaming have much in common, since both simply allow all the actions of the underlying process but change some of

the names of the events. Any event not being hidden retains its own name under $\setminus B$, but when this event is $\checkmark$ we need a separate rule to respect our convention that the result process is always then Ω.

$$\frac{P \xrightarrow{x} P'}{P \setminus B \xrightarrow{x} P' \setminus B} \quad (x \notin B \cup \{\checkmark\}) \qquad \frac{P \xrightarrow{\checkmark} P'}{P \setminus B \xrightarrow{\checkmark} \Omega}$$

Events in B are, on the other hand, mapped to τ.

$$\frac{P \xrightarrow{a} P'}{P \setminus B \xrightarrow{\tau} P' \setminus B} \quad (a \in B)$$

Renaming has no effect on either τ or $\checkmark$ actions:

$$\frac{P \xrightarrow{\tau} P'}{P[\![R]\!] \xrightarrow{\tau} P'[\![R]\!]} \qquad \frac{P \xrightarrow{\checkmark} P'}{P[\![R]\!] \xrightarrow{\checkmark} \Omega}$$

Other actions are simply acted on by the renaming:

$$\frac{P \xrightarrow{a} P'}{P[\![R]\!] \xrightarrow{b} P'[\![R]\!]} \quad (a\ R\ b)$$

We here give the semantics of just one parallel operator, since others can be deduced from it: $P \underset{X}{\parallel} Q$ synchronises P and Q on all actions in X, lets them communicate freely on other events, and terminates when they both have. Both arguments are active

$$\frac{P \xrightarrow{\tau} P'}{P \underset{X}{\parallel} Q \xrightarrow{\tau} P' \underset{X}{\parallel} Q} \qquad \frac{Q \xrightarrow{\tau} Q'}{P \underset{X}{\parallel} Q \xrightarrow{\tau} P \underset{X}{\parallel} Q'}$$

There are three rules for ordinary visible events: two symmetric ones for $a \notin X$

$$\frac{P \xrightarrow{a} P'}{P \underset{X}{\parallel} Q \xrightarrow{a} P' \underset{X}{\parallel} Q} \quad (a \in \Sigma \setminus X)$$

$$\frac{Q \xrightarrow{a} Q'}{P \underset{X}{\parallel} Q \xrightarrow{a} P \underset{X}{\parallel} Q'} \quad (a \in \Sigma \setminus X)$$

and one to show $a \in X$ requiring both participants to synchronize

$$\frac{P \xrightarrow{a} P' \quad Q \xrightarrow{a} Q'}{P \underset{X}{\parallel} Q \xrightarrow{a} P' \underset{X}{\parallel} Q'} \quad (a \in X)$$

The parallel operator terminates when both its arguments have. There are two approaches to this. The simple one is to assume that ✓s can be synchronised between the processes, but this contradicts the idea that ✓ is an uncontrollable signal. Therefore we have the operator watch its arguments and terminate when both its arguments have:

The terminations of the two arguments are turned into τ's much as in the first argument of $P;\ Q$.

$$\frac{P \xrightarrow{\checkmark} P'}{P \underset{X}{\parallel} Q \xrightarrow{\tau} \Omega \underset{X}{\parallel} Q} \qquad \frac{Q \xrightarrow{\checkmark} Q'}{P \underset{X}{\parallel} Q \xrightarrow{\tau} P \underset{X}{\parallel} \Omega}$$

Once one of its arguments has terminated and become Ω, all the rules above for $\underset{X}{\parallel}$ still apply, bearing mind that Ω itself has no transitions (being basically equivalent to $STOP$) so that $P \underset{X}{\parallel} \Omega$ can only do those of P's actions not in X. After the second argument has terminated the composition will have become $\Omega \underset{X}{\parallel} \Omega$: it can now terminate using the following rule.

$$\frac{}{\Omega \underset{X}{\parallel} \Omega \xrightarrow{\checkmark} \Omega}$$

Another view of the above rules for parallel termination is that we should not regard Ω as a real observable process, but think of $P \underset{X}{\parallel} \Omega$, $\Omega \underset{X}{\parallel} Q$ and $\Omega \underset{X}{\parallel} \Omega$ as a suggestive notation for two unary operators (on Q and P respectively) and a constant, which happens to be equivalent to $SKIP$. The latter is consistent with the view, which we will discuss later, that when a process terminates it should disappear from whatever term succeeds it.

Other forms of CSP parallel are *interleaving* $P \ ||| \ Q$, equivalent to $P \underset{\emptyset}{\parallel} Q$, and *alphabetised parallel* $P \ {}_A\|_B \ Q$ which forces P to communicate all events in B, and Q in C. Provided that P and Q do not attempt to communicate outside A and B respectively it is equivalent to $P \underset{A \cap B}{\parallel} Q$.

CSP provides two ways of getting one process to take over from another without the first one actually terminating: interrupt $P \triangle Q$ allows P to run, but at any time offers the initial events of Q. If one of the latter happens then Q takes over. Both arguments are initially active.

$$\frac{P \xrightarrow{\tau} P'}{P \triangle Q \xrightarrow{\tau} P' \triangle Q} \qquad \frac{Q \xrightarrow{\tau} Q'}{P \triangle Q \xrightarrow{\tau} P \triangle Q'}$$

If P terminates, so does the whole

$$\frac{P \xrightarrow{\checkmark} P'}{P \triangle Q \xrightarrow{\checkmark} \Omega}$$

If P performs $a \in \Sigma$, then the possibility of interruption remains

$$\frac{P \xrightarrow{a} P'}{P \triangle Q \xrightarrow{a} P' \triangle Q}(a \in \Sigma)$$

If Q performs $a \in \Sigma \cup \{\checkmark\}$, then it takes over as with $\Box$:

$$\frac{Q \xrightarrow{a} Q'}{P \triangle Q \xrightarrow{a} Q'}(a \in \Sigma \cup \{\checkmark\})$$

The other is an operator which allows an event from P in the set A to close it down and hand over to Q: the *throw* operator $P \Theta_A Q$. Only the first argument is active:

$$\frac{P \xrightarrow{\tau} P'}{P \Theta_A Q \xrightarrow{\tau} P' \Theta_A Q}$$

If it terminates, so does the operator

$$\frac{P \xrightarrow{\checkmark} P'}{P \Theta_A Q \xrightarrow{\checkmark} \Omega}$$

It is allowed to perform $a \notin A$ and carry on

$$\frac{P \xrightarrow{a} P'}{P \Theta_A Q \xrightarrow{a} P' \Theta_A Q}(a \notin A)$$

whereas $a \in A$ hands control to Q:

$$\frac{P \xrightarrow{a} P'}{P \Theta_A Q \xrightarrow{a} Q}(a \in A)$$

This completes our introduction to the language by the means of SOS combinator operational semantics, except that we have not given the semantics for the priority operator. This can be found in Section 2.4.

2.3 CSP-like operational semantics

Note that all of the conditions (premises) above the line in the rules we have seen are *positive* in the sense defined earlier.

They also have the following properties

- If an argument process performs an action $P \xrightarrow{x} P'$ in the premises, and remains alive after the action that is being derived, then P has become P' in the result.

- If an argument process has not performed an action in the premises, then if it remains alive after the derived action it stays in its initial state.

- If the argument process P appears in any of the premises of the operator $F(P, \ldots)$ (i.e., the initial actions of $F(P, \ldots)$ depend on those of P), and $P \xrightarrow{\tau} P'$, then $F(P, \ldots) \xrightarrow{\tau} F(P', \ldots)$. In other words, if an argument of F is active in the sense discussed in defining the SOS operational semantics above so that F can use its actions immediately[4], then P can perform a τ without otherwise changing the state of $F(P, \ldots)$. We can regard this as F simply letting such τs happen unobserved even by it. There are no other rules with τ as a premise.

- Under the same conditions, if P can perform $\checkmark$ then $F(P)$ can *either* perform $\checkmark$ to the special term Ω *or* perform τ to a term not involving P.

- No argument process ever appears twice or more in the result of any actions. Thus there are no rules such as

$$\frac{P \xrightarrow{a} Q}{F(P) \xrightarrow{a} G(Q, Q)}$$

  This is the *no cloning* property. This can in fact be breached in a way by recursion: it is permissible to have recursions such as

$$H(Q) = (a \rightarrow H(Q)) \underset{A}{\parallel} Q$$

  which replicates Q. However, this cloning only replicates Q in its original, unstarted form. CSP contains no way, recursive or otherwise,

[4]For technical reasons it may occasionally be necessary to count an argument as active even though the operator makes no direct use of its visible actions: see Section 2.3.

of having a process P active (i.e., perhaps having performed actions already) and then cloning it into two copies which can in any way be run side-by-side or compared. This is hugely important to the style of model and algebraic laws normally used for CSP, specifically ones based on *linear behaviours* and including distribution properties over nondeterministic choice.

Thus, though no standard CSP operator can copy any argument, operators definable in CSP via recursion, like $H(\cdot)$ above, can make arbitrarily many copies of a fixed, as yet unstarted, argument.

Note that the first and last principles here imply that no argument process ever performs two or more actions, whether in sequence or independently, in the premises of a single action.

In [27, 26], the author codified all of the above conditions together, including the banning of negative premises, and described an operational semantics all of whose operators obey these principles as *CSP-like*. The clearest way of doing this was creating a new notation for operational semantics that is so constrained that it can only express CSP-like operators. This was termed *combinator* operational semantics.

Before defining combinator operational semantics we remark the concept of CSP-like operational semantics bears close comparison with *simply WB cool* rules as defined in [11]. This is a restriction on SOS designed to ensure that operators defined respect weak bisimulation (hence WB) in the same way that we are aiming at CSP equivalences. We will perform a proper comparison in Section 6, but for now remark that CSP-like is stronger than simply WB cool by the addition of the no-cloning condition and other closely related ones. In the present paper we adopt some of the nomenclature of [11], even though this is different from that in [27, 26]. Specifically an argument whose behaviour contributes to the first-step behaviour of an operator is termed *active*, and *inactive* otherwise. [26, 27] termed these **on** and **off** respectively. The rules which simply promote a τ action are termed *patience* rules.

We define this form of semantics here so that we can later extend it to handle negative premises.

In giving a combinator operational semantics for the operator $F(P_1, \ldots, P_n)$, the first thing we need to identify is which of the P_i are initially active: which of them appear in the premises of the SOS operational rules. The notation we will use for an operator with active arguments $\mathbf{P}$ and inactive ones $\mathbf{Q}$ in defining its combinator semantics will take the form $F^{\mathbf{Q}}(\mathbf{P})$, emphasising that the active ones are those that immediately relevant. (This is again

different from [26, 27], the motivation being to make a greater distinction between the immediately relevant active arguments, on which the combinator acts, and ones with a delayed role.) In general we allow an infinite number of components to $\mathbf{Q}$, which we treat as a function from some indexing set I to processes. This case does arise in CSP, both thanks to taking the nondeterministic choice of an infinite number of processes and, in the case where the alphabet Σ is infinite, prefix constructs (such as $c?x \rightarrow \cdot$ when the type of c is infinite). However we only allow finitely many active arguments since not only does the infinite case not arise in CSP, but it would be theoretically problematic thanks to patience rules. In the definition below we will suppose there are m active arguments.

Based on the operational semantics of CSP given above:

- All arguments of prefixing and nondeterministic choice are initially inactive: the initial actions of these constructs depend only on the construct itself. In each case the effect of such an initial action is to activate one of the arguments.

- All arguments of external choice, parallel operators, hiding, renaming and interrupt are active.

- The first arguments of $P;\ Q$ and $P\,\Theta_A\,Q$ are active, and the second inactive because their actions only become relevant after the respective first argument has performed at least one. We get the same pattern for sliding choice $P \rhd Q$, but this time because Q only starts up after a τ introduced by the operator itself independently of P.

Like SOS, a combinator operational semantics takes the form of a collection of rule schemas, with events, sets of events etc varying under side conditions to create sets of rules for individual operators. An individual rule takes the form of a triple, sometimes abbreviated to a pair.

- The first component is a tuple with one component for each active argument. The members of this m-tuple $(x_1, \ldots, x_m)$ are taken from $\Sigma \cup \{\checkmark, \cdot\}$. The meaning of this tuple is that all active arguments whose component is not "$\cdot$" perform the relevant action, in a synchronised fashion, for the rule to fire. (We will often put quotes like this around $\cdot$ in text to help distinguish it.) Note that in some CSP operators $m = 0$, which simply says that all of the operator's actions are unconditional on arguments' actions. In these cases we write the now null premises as $^-$. There are special conditions on tuples with $\checkmark$ actions, which we

will discuss below. Note that τ is not permitted in these tuples: we will discuss this below.

- The second component is an action y in $\Sigma \cup \{\checkmark, \tau\}$ which represents the result action of the rule: the one that the operator performs when the active arguments perform the components of the first. The CSP hiding operator gives a case where a visible action is turned into τ, hence the possibility of y being τ. Thus we do not allow our operator to observe the τs that active arguments perform as x_i, but do allow it to generate its own τs or turn a synchronisation of argument actions into τ.

- The third component represents the syntax that the process becomes after the action. There are three possibilities here:

  1. The result of the action does not change the process's shape: it is still the same operator applied to the same arguments, the only change being that those active arguments that have participated in the action have moved forward according to respective component actions. This is a common case, and applies to all actions not involving $\checkmark$s of parallel, hiding and renaming operators, and combinations of these. In this case the third component is omitted, so the combinator becomes a pair. We will term such combinators *homogeneous*.
  2. The result action is $\checkmark$, Since, by convention, the result action of $\checkmark$ is always Ω, there is no need to record it, and so the final component is again omitted.
  3. In any other case we do need to record the state that the process moves into. This will always be a piece of syntax with placeholders for the active and inactive arguments. The form of this syntax has to be restricted so as to prevent either the cloning or suspension of the active arguments of the original operator. The syntax can, however, do what it likes with the inactive arguments, and discard any argument it likes.

     We discuss the allowable forms of syntax below.

It was observed by Hoare that one of the properties that most characterises CSP is distributivity over nondeterministic choice: all non-recursive CSP operators have, in each argument individually, the property that $F(P \sqcap Q) = F(P) \sqcap F(Q)$. It has long been recognised that this is intimately connected to the no-cloning property of its operational semantics, because if

F's argument could be cloned, it is likely that we could tell the difference between the behaviour of $F(P \sqcap Q)$, in which the choice between P and Q can be delayed until after cloning by F, and $F(P) \sqcap F(Q)$, where it can not. This is because F running one copy of P and one or Q could easily show a behaviour that it cannot when running one copy of each.[5]

Notice here that we make no restrictions about how many times an inactive argument of F can be used. This is perhaps as well, since we should observe that constructs such as $H(P)$ defined earlier copy inactive arguments.[6]

The way we want combinators to build the syntax of successor processes can be defined by specifying that they must treat active arguments, if they are retained at all, in a way that keeps them active and follows the principles of distributivity.

This is a piece of syntax T (made up from CSP and CSP-like constructs) in which each argument (active and inactive) is represented by some standardised identifier. For us these are bold-face indices drawn from $\{1, \ldots, m\} \cup I$, so $\mathbf{1}$ represents the first active argument, and so on. The result state is now T with the substitutions:

- An index $\mathbf{i} \in \{1, \ldots, m\}$ is replaced by P_i or P_i' such that $P_i \xrightarrow{x_i} P_i'$ depending on whether $x_i = \cdot$ or $x_i \in \Sigma$. If x_i is $\checkmark$ then $\mathbf{i}$ may not appear in T.

- An index $\mathbf{i} \in I$ is replaced by Q_i.

To follow the principles above we have to impose conditions on T:

- No active index $\mathbf{i} \in \{1 \ldots, m\}$ can appear more than once in T.

- Such active indexes only appear at *immediately distributive* (ID) places in T, (i.e., where the operational semantics we can derive for T makes a process argument placed here initially active). This is easy to define by structural recursion:

[5]For example, if $F(P)$ clones P and interleaves it with itself, then $F(a \rightarrow STOP \sqcap b \rightarrow STOP)$ could perform the trace $\langle a, b \rangle$, which is not possible for $F(a \rightarrow STOP)$ or $F(b \rightarrow STOP)$.

[6]For completeness we might note that it is also possible to define operators in CSP, such as $G(P) = P \,|||\, P \,|||\, (a \rightarrow G(P))$ where it is possible for an argument to be initially active several times over and also be present in an inactive form. We do not provide direct support for this type of construct in combinator operational semantics, but note that if one did desire to use one, one could equally define an operator in which all active instances, and a single inactive one, were separate, and apply this to multiple instances of the same process. Here it would be $G'(P, P, P)$, where $G'(P, Q, R) = P \,|||\, Q \,|||\, (a \rightarrow G'(R, R, R))$.

– The appearance of **i** in the simple term **i** is ID.
– If **i** appears ID in the term T, then it appears ID in $\bigoplus(\ldots, T, \ldots)$, where the place T occurs at an **active** place in the arguments of the CSP-like operator $\bigoplus$.
– No other appearance of **i**, including any in a recursive definition, is ID.

It is important to note that once an argument is active, it must stay active as long as it is present in the term. In CSP terms this is because CSP provides no mechanism that can suspend and re-start a process, at least in terms of its ability to perform τ actions. (One can use a regulator to stop Σ actions and then take away the restriction, but there is no way of stopping it performing τs in the same way other than throwing it away permanently). Also, if a process were suspended, becoming inactive, copying it would breach the no-cloning principle. There is therefore no way in which we can write a rule in which Q performing an action leads it to the state $P;\ Q'$.

However we could do this if we had an alternative version of the sequential composition operator in which both arguments count as active. It is for this reason that we might occasionally want an operator where an argument is counted as active even though its visible actions are not immediately relevant.

It is worth remarking that the pieces of syntax T above can contain arbitrary *closed* CSP processes at any point without restriction[7]. In other words, either the whole expression or any argument to any operator can be any process term that does not depend on process variables representing arguments or anything else.

Drawing from examples already seen above, the hiding operator $P \setminus X$ has rules $(a, a)[a \notin X]$ and $(a, \tau)[a \in X]$, using the convention that for operators like this one with a single active argument, we write a rather than (a) for a tuple of actions from each. In this case the result of $P \setminus X$ processing an action $P \xrightarrow{a} P'$ is always $P' \setminus X$, so we can use the combinator form without a result process. On the other hand, the resolution of $P \,\Box\, Q$ does change the process structure, so its rules are

$$((a, \cdot), a, \mathbf{1}) \qquad \text{and} \qquad ((\cdot, a), a, \mathbf{2})$$

[7]In fact it is easy to show that any LTS at all is the operational semantics of such a process, using a straightforward potentially infinite mutual recursion and hiding to create any τs.

indicating that either side can perform any action in Σ which resolves the choice, eliminating the $\Box$. These rules have exactly the same meaning as the SOS ones for Σ actions in $\Box$ given above.

Comparing these rules against the SOS operational semantics given earlier, there are two omissions: the rules using active arguments' τs and $\checkmark$s. The first of these is very simple to solve: there is no need to write down patience rules (any τ by an active argument becomes a τ of the operator without changing the state other than progressing the argument to its post-τ state) since they *always* apply. Thus any specification of a combinator operational semantics assumes that patience rules apply without the need to write them down.

The rules with a $\checkmark$ premise that generate the event $\checkmark$ were dealt with above as one of the cases where no third component is necessary. In fact the rules for hiding and $\Box$ all fall into this category: $(\checkmark, \checkmark)$ for hiding and $((\checkmark, \cdot), \checkmark)$ plus $((\cdot, \checkmark), \checkmark)$ for $\Box$.

We present $\checkmark$ rules in a very similar format, but where necessarily the tuple of actions upon which the rule depends contains one $\checkmark$ and all other components are "$\cdot$". All such rules whose result action is $\checkmark$ appear as a pair, following the above convention, for example the $\checkmark$ rules of $P \Box Q$ seen above.

Recall that we stated above that a process that contributes a $\checkmark$ to an action is in any case not present in the successor process. This is naturally because the operator knows that the said argument is finished. We make some more stipulations about the role of $\checkmark$, which are present to implement the assumption that $\checkmark$ is an observable action but not one that can be controlled by external observers or by extension the operator itself.

- If a component of the tuple of premises is $\checkmark$, it is the only non-"$\cdot$" component. *This says that $\checkmark$s can't be synchronised with other actions, even other $\checkmark$s, because synchronisation implies the ability to delay until partner actions are available.*

- The only result actions y possible from such a tuple are τ and $\checkmark$. *We cannot turn an uncontrollable action into a controllable one, since doing so implies we would have to control the $\checkmark$ when obliged to control the Σ-event it contributes to.* Thus we can only have the third component T in the case where $y = \tau$, which represents the case where one of the active arguments has closed down but there is still more for the process to do. The most obvious instance of this is in sequential composition, where a $\checkmark$ of the first argument becomes τ and starts up the second. identifier representing the second argument, and $a \rightarrow P$ has

the combinator $(^{-}, a, \mathbf{p})$ where $\mathbf{p}$ is the index of the inactive argument P.

A good example to study is the distributed termination of $P \underset{X}{\parallel} Q$: note that the SOS semantics given earlier are consistent with the above provided we take the approach that a process in parallel with Ω is a unary operator on it, needing separate combinators.

Thus $P;\ Q$ has the combinator $(\checkmark, \tau, \mathbf{q})$, where $\mathbf{q}$ is the index to the inactive second argument.

Complete combinator operational semantics for CSP can be found in [26].

Definition 1 *An operator (language) is CSP-like if and only if it (all its operators) can be given a combinator operational semantics.*

CSP, naturally, is CSP-like but CCS[16] is not because of the way in which τ resolves $+$. However the version of π-calculus [17] presented in [29] can be shown to be CSP-like [25] because of the way it localises the use of $+$.

The justification for this definition is the following theorem.

Theorem 1 *Every CSP-like operator F has a translation to CSP which we write F_{CSP} such that, for any collection of arguments $(\mathbf{P}, \mathbf{Q})$, the operational semantics of $F^{\mathbf{Q}'}(\mathbf{P}')$; SKIP and $F^{\mathbf{Q}}_{CSP}(\mathbf{P})$ are strongly bisimilar, where $\mathbf{P}'$ and $\mathbf{Q}'$ are formed by replacing each component P by the semantically equivalent process P; SKIP.*

The reason for this slight alteration to the arguments and result with *SKIP* is that CSP does not contain the mechanisms for marshalling $\checkmark$ events that it does for others. Note that if P is a process that never performs $\checkmark$ then P and $P;\ SKIP$ are themselves strongly bisimilar, so that if none of the arguments terminates the transformation from $(\mathbf{P}, \mathbf{Q})$ to $(\mathbf{P}', \mathbf{Q}')$ is to all intents and purposes the identity.

Therefore any CSP-like operator has a fully compositional semantics over any model of CSP.

The proof of this result can be found in [27, 26]. Though not quite all the cases are covered, the proof of the main theorem of the present paper provides a model for a slightly different proof of Theorem 1, if the bits about negative premises are removed.

Much to the author's surprise, Tom Gibson-Robinson (then a final year undergraduate) actually implemented the translation from CSP-like F to F_{CSP} in a tool called TYGER [9].

2.4 Priority

While there have been a number of versions of CSP with priority, for example [15, 14, 8] the one we use in this paper is that introduced in [26]. This is conceptually simple, because it does not require any re-interpretation of LTS's or CSP models as entities where one action has priority over another. Instead $\mathbf{Pri}_{\leq}(P)$ inputs an ordinary LTS and the result is another ordinary one. This is an operator whose definition is most naturally cast in terms of operational semantics, though these are definitely not CSP-like. The parameter $\leq$ is a partial order on events $\Sigma \cup \{\tau, \checkmark\}$ which is subject to several conditions that we will state below, The SOS operational semantics are easy to state

$$\frac{P \xrightarrow{x} P' \wedge \forall\, y.y > x \Rightarrow \neg P \xrightarrow{y}}{\mathbf{Pri}_{\leq}(P) \xrightarrow{x} \mathbf{Pri}_{\leq}(P')}$$

In other words P performs actions that are not strictly lower under $\leq$ than some other action that P can perform from the same state. In the above, x and y range over the whole of $\Sigma \cup \{\tau, \checkmark\}$.[8]

Though the above definition makes perfect sense operationally whatever order is used, in order to make it consistent with the basic tenets of CSP we need to respect the ideas that τ and $\checkmark$ are not controllable and that every process is equivalent to the one where a single τ precedes it. To protect these properties we insist that

- τ and $\checkmark$ are both maximal in $\leq$: they are not dominated by any other event.
- If $a < b$ for any actions a and b, then $a < \tau$ and $a < \checkmark$.

The need for these conditions is further explained in [26].

Even with these restrictions, this is not a CSP-like operator because it has negative premises in its operational semantics. This is emphasised by the fact that it is not compositional over most established CSP models such as traces and failures-divergences. For example the processes $(a \rightarrow STOP) \sqcap (b \rightarrow STOP)$ and $(a \rightarrow STOP) \,\Box\, (b \rightarrow STOP)$ have the same traces semantics, but any $\leq$ where $a < b$ leaves the former process unchanged but turns the latter into one equivalent to $b \rightarrow STOP$.

It is only the richest CSP models that give $\mathbf{Pri}_{\leq}(P)$ any hope of being compositional. Of those discussed in Chapters 10, 11 and 12 of [26], the only

[8] Earlier we stated that the order $\leq$ should not have any infinite ascending chains. This is because if $a_1 < a_2 < a_3 < \ldots$, then the process $\mathbf{Pri}_{\leq}(\{\Box\{a_i \rightarrow STOP \mid i \in \mathbb{N}\})$ could not perform any action at all, which seems unnatural.

models which guarantee compositionality for the full range of permitted $\leq$ are the $\mathcal{FL}$ class of models, whose basic behaviours are traces extended by one of the following before each event and after the last:

- The symbol $\bullet$, indicating that the state from which the next event happened, or which applies at the end of the trace, has not been observed to be stable (i.e., a state where no τ or $\checkmark$ is possible).
- Where stability has been observed, the exact set of events that the state offers.

Thus a typical behaviour looks like $\langle A_0, a_1, A_1, \ldots, A_{n-1}, b_n, A_n\rangle$ with the b_i being drawn from Σ. and the A_i being drawn from $\{\bullet\} \cup \mathcal{P}(\Sigma)$ (The construction of the model assumes that the observer has no obligation to observe stability even when the process is stable, therefore for any recorded behaviour with a proper acceptance set, there is another which is the same except for replacing that set by $\bullet$.) Additionally there are behaviours of the form $\langle A_0, a_1, A_1, \ldots, A_{n-1}, b_n, \bullet, \checkmark\rangle$, meaning that any $\checkmark$ is the final thing observed, and the state preceding it cannot be stable.

The semantics of $\mathbf{Pri}_{\leq}(P)$ over this model are as follows, quoted from [28] and extended to cover the case of $\checkmark$:

With respect to $\mathcal{FL}$, the semantics of $\mathbf{Pri}_{\leq}(P)$ are the behaviours :

$$\begin{array}{c}\{\langle A_0, b_1, A_1, \ldots, A_{n-1}, b_n, A_n\rangle \mid \langle Z_0, b_1, Z_1, \ldots, Z_{n-1}, b_n, Z_n\rangle \in P\} \\ \cup \\ \{\langle A_0, b_1, A_1, \ldots, A_{n-1}, b_n, \bullet, \checkmark\rangle \mid \langle Z_0, b_1, Z_1, \ldots, Z_{n-1}, b_n, \bullet, \checkmark\rangle \in P\}\end{array}$$

where for each i one of the following holds:

- b_i is maximal under $\leq$ and $A_{i-1} = \bullet$ (so there is no condition on Z_{i-1} except that it exists).
- b_i is not maximal under $\leq$ and $A_{i-1} = \bullet$ and Z_{i-1} is not $\bullet$ and neither does Z_{i-1} contain any $c > b_i$.
- Neither A_i nor Z_i is $\bullet$, and $A_i = \{a \in Z_i \mid \neg\, \exists\, b \in Z_i . b > a\}$,
- and in each case where $A_{i-1} \neq \bullet$, $b_i \in A_{i-1}$.

In [26] it is incorrectly stated that $\mathbf{Pri}_{\leq}(P)$ is compositional in refusal testing models (see [18, 26]). This is true for restricted classes of $\leq$ as demonstrated in [28], specifically ones with no three members of Σ such that $c < a$, a and b are incomparable but $c \not< b$. However since the uses

of priority we will see in this paper will frequently have such triples, we will disregard refusal testing models in this paper.

FDR implements a *prioritise* operator which is the same as the one presented here, except that it restricts the partial order to ones presentable as a series of subsets of Σ with successively lower priority, with all events not included in these sets being incomparable to all of them as well as $\{\tau, \checkmark\}$. One does not lose any generality through this restriction, since $\mathbf{Pri}_{\leq}(P)$ for any allowable partial order can be generated (at least for finite alphabets) by one or more nested instances of the FDR version. The reason for *prioritise* taking this different form is that it is a lot simpler than requiring the programmer to create a suitable representation of a general member of our restricted class of partial orders. At the time of writing we are discussing adding a full version of $\mathbf{Pri}_{\leq}(\cdot)$ as an advanced alternative to *prioritise*.

We remarked earlier that priority posed an additional challenge in the presence of under-defined recursions. We can now demonstrate this. Contemplate the definition $\mu\, p.\mathbf{Pri}_{\leq}(p[\![^b/_a]\!] \,\Box\, a \rightarrow STOP)$, where $a < b$. Under recursion rule (A), given that we have specified $a < \tau$, this definition simply delivers an infinite stream of τs. Under (B) it is *contradictory*, since no τs now appear, and p can perform an a if and only if it *cannot* perform a b, but it can perform b if and only if it *can* perform a.

Since priority is not a CSP-like operator, we really need a new name for the language that includes it: we will call it *Pri-CSP*.

It is interesting to note that *prioritise* is implemented in FDR as a separate operator on LTSs. It cannot be folded into the main implementation of CSP in FDR, via supercombinators (described, for example, in [26]), which is no surprise because there is a very close relation between supercombinators and the combinator operation semantics which we know *prioritise* does not have.

3 What can we express in Pri-CSP?

Having demonstrated a strong expressibility result for CSP, the question arises as to whether we can find a similar characterisation when we add our priority operator. This operator shares some of the qualities of CSP-like operators, for example

- It has the patience property, since τ actions are never blocked.
- It cannot clone its argument.

Indeed the only one it obviously does not have is the use of negative premises in its SOS operational semantics. It is tempting to hope that we can simply extend Theorem 1 to encompass operational semantics that do have negative premises, and the proof of a version of this is our main result.

To characterise what can be expressed in Pri-CSP we extend the expressive power of combinator operational semantics to encompass negative premises. Recall that the first component of a combinator, representing its premises, is a tuple of actions from the active processes. We can extend this by turning the components of this tuple into pairs. The first component is either an action in Σ that the corresponding process should perform or "$\cdot$" if it does not perform one in the action. The second component is a set of events, which if non-empty contains τ and $\checkmark$ (if not written down they are assumed implicitly), that the process must not be able to perform if the rule is to fire. To suggest its meaning, we annotate such sets of negative premises S with the negation symbol $\neg$, so it is written $\neg S$. Any negative premise requires that the corresponding argument process be stable before it can be satisfied.

We will be liberal with the way we write down such pairs: where one or other component is trivial (i.e., $\cdot$ or $\emptyset$ (rather than $\{\checkmark, \tau\}$)) we will just write the other, and if both are trivial we will just write "$\cdot$".

There is no difference in the structure of the second component of combinators. However issues that we will discuss in Section 5.1 mean that we choose to be a lot more restrictive in the syntax of the allowed third component syntax T. Specifically we restrict the third component to be any one of

- One of the argument processes by itself (a common case in CSP): this can be an inactive or inactive one in the original state.
- Any constant CSP process (one that does not refer to any argument)
- Any Pri-CSP operator application where each active argument of the original operator, if it appears at all, appears in exactly one place amongst the active arguments of the new operator.

This restriction applies to all the combinators (even those with no negative premises) in any operational semantics involving any combinator involving negative premises. In practice this is not a huge restriction, since in every operator semantics the author is aware of, the above restrictions hold.

There is again the assumption of a patience rule for each active argument, and a *homogeneous* n-combinator is one in which the third component is omitted because the result has the same structure as the initial process.

Exactly the same restrictions apply as before to rules with ✓ events as premises, with the additional restriction that no negative premises are permitted in such rules.

Any such combinator that has a negative premise will be termed an *n-combinator*, and an *n-combinator* operational semantics is one in terms of these and ordinary combinators. A *positive* combinator semantics is one with only ordinary combinators.

$\mathbf{Pri}_{\leq}(\cdot)$'s operational semantics can itself be expressed in n-combinators, noting that it has a single active argument: if has the ✓ rule $(\checkmark, \checkmark)$, the implicit patience rule and, for each $a \in \Sigma$ maximal in $\leq$ the simple combinator (a, a). For non-maximal a it has the n-combinator $((a, \neg\{x \in \Sigma \cup \{\tau, \checkmark\} \mid a < x\}), a)$, where we note that the set of negative premises always includes τ and ✓ thanks to the restrictions placed on $\leq$ in the definition of the priority operator, which mean that the similar restrictions we placed on negative premises are also satisfied.

Recall that the operator $P \,\Theta_A\, Q$ closes down P and starts Q whenever P communicates an element in A. We can think of this as P throwing an exception. With n-combinators we could build an operator in which any deadlock in P was caught and starts Q: with the active argument P it would simply need the combinators (a, a) for $a \in \alpha P$ and $(\checkmark, \checkmark)$, plus the n-combinator $(\neg\alpha P, \tau, \mathbf{q})$ where $\mathbf{q}$ points to the inactive argument Q. Once we have discussed the implementation of general negative premises later, it will be obvious how this construct can be simulated in Pri-CSP.

As another example, consider an "angelic choice" operator which behaves like $P \,\Box\, Q$ except that it does not behave nondeterministically when P and Q have the same initial action. Rather both perform the action and wait for the choice to be resolved later. The author knows two distinct operational semantics for versions of this operator. The first is a complex one $\boxdot_P$ with a completely positive operational semantics in which when one of the argument performs an event, the other is allowed to follow it, and subsequent events later. Its combinator operational semantics is given in detail in Chapter 9 of [26], and comes as one of an infinite family of operators in representing the case where either argument is an arbitrary non-terminated trace ahead. Unless the two processes have performed the same trace, the one that is behind is only allowed to perform τ (via the patience rule), ✓ (which becomes an external τ and resolves the choice in favour of the other process), or the first catch-up event (which becomes an external τ and reduces the deficit.)

There is a simpler and more natural definition $\boxdot_N$ that involves negative premises, and defines a subtly different operator both at the levels of

operational semantics and abstract ones. This simply says that when one of P and Q can perform an event in Σ that the other cannot, this resolves the choice. The n-combinators for this version are

- $((\checkmark, \cdot), \checkmark)$ and $((\cdot, \checkmark), \checkmark)$: if either argument terminates then so does the combination.
- $((a, a), a)$ for each $a \in \Sigma$: if both arguments perform the same Σ-action then they do so lockstep and the choice between them is delayed.
- $((a, \neg\{a, \tau, \checkmark\}), a, \mathbf{1})$ $((\neg\{a, \tau, \checkmark\}, a), a, \mathbf{2})$ for $a \in \Sigma$: if one argument can perform an action but the other can neither do it nor $\checkmark$ nor τ, then it resolves the choice.

The biggest difference between the two versions shows up when one of the arguments is divergent. Every trace of P is one of $P \boxdot_P \mathbf{div}$ (**div** being the simply divergent process alluded to earlier), but $P \boxdot_N \mathbf{div}$ only has the empty trace unless P has the trace $\langle\checkmark\rangle$, in which case it has that too: P cannot proceed with $a \in \Sigma$ because **div** can always perform a τ.

It is not immediately obvious that $\boxdot_N$ can be implemented in Pri-CSP, but it can be done. Ignoring termination, which makes it more complex, we can achieve this by enlarging Σ by further copies a_1, a_2, a_3 and a_4 of each a in the original Σ, which we will now refer to as Σ_0, with Σ_i for $i > 0$ being the copies. The implementation of $P \boxdot_N Q$ is then

$$(((\mathbf{Pri}_{\leq_1}((P[\![R']\!] \;|||\; RUN_{\Sigma_2})) \underset{\Sigma_0 \cup \Sigma_1 \cup \Sigma_2}{\|} \mathbf{Pri}_{\leq_2}(Q[\![R'']\!] \;|||\; RUN_{\Sigma_1})) \underset{\Sigma}{\|} Reg))[\![CR]\!]$$

where

- R' maps each $a \in \Sigma_0$ to a, a_1 and a_3
- R'' maps each $a \in \Sigma_0$ to a, a_2 and a_4
- CR maps each a and a_i to a.
- Reg initially permits any event in $\Sigma_0 \cup \Sigma_1 \cup \Sigma_2$, but once an a_1 occurs then only members of Σ_3 may occur thereafter, and likewise once an a_2 occurs then only members of Σ_4 are allowed thereafter.
- $\leq_1$ places all a_2 events below all members of Σ_0.
- $\leq_2$ places all a_1 events below all members of Σ_0.

The interpretation of these events is:

- a is the event representing joint progress by the two processes, leaving the choice unresolved.
- a_1 is an event which P performs when Q cannot perform the same one.
- a_2 is an event which Q performs when P cannot perform the same one.
- a_3, unsynchronised with Q, is an event that P can perform when the choice has already been resolved in its favour.
- a_4, unsynchronised with P, is an event that Q can perform when the choice has already been resolved in its favour.

You can regard a_2 as performed by $\mathbf{Pri}_{\leq_1}(P[\![R']\!] \,|||\, RUN_{\Sigma_2})$ as saying that P cannot perform the corresponding a thanks to $\leq_1$. Thus when Q synchronises a_2 with this process it means that the choice has been resolved in Q's favour. In understanding the simulation's behaviour when in one of these two modes it is important to note that, whichever of P and Q has been left behind in favour of the other, is necessarily in a stable state (if not an a_1 or a_2 could not have happened), so blocking its visible actions means it does nothing at all thereafter.

This construction[9] was far from simple, but does begin to show how an operator which conceptually has nothing to do with priority can be implemented using it together with the rest of CSP.

This construction motivates the following definition:

Definition 2 *An operator has* Pri-CSP-like *operational semantics if its operational semantics can be given according to the above conventions in terms of combinators and n-combinators.*

We will show in the next section that all Pri-CSP-like operators are implementable in Pri-CSP.

In Section 5 we will show how to liberalise the definition of Pri-CSP-like to encompass yet further types of premise.

[9]It is interesting to note that using it, $P \boxdot_N Q$ is always a finite-state process if P and Q are, something that is not true of $P \boxdot_P Q$. If P is any process with an infinite trace then $P \boxdot_P Q$ is infinite state.

4 Expressibility theorem

The following is the central result of this paper.

Theorem 2 *Suppose the operator $F^{\mathbf{Q}}(\mathbf{P})$ is Pri-CSP-like together with all other operators reachable (transitively) through the T third components of its combinators. Then for any arguments $\mathbf{P}$ and $\mathbf{Q}$, $F^{\mathbf{Q}}(\mathbf{P})$ is expressible in Pri-CSP in the sense that the simulation is strongly bisimilar to $F^{\mathbf{Q}'}(\mathbf{P}')$; SKIP where $\mathbf{P}'$ and $\mathbf{Q}'$ are defined as previously by replacing each component P by P; SKIP.*

This of course implies that such operators have a compositional semantics over the $\mathcal{FL}$ family of models.

As in [27, 26], our proof will be to construct the CSP representation. This is, of course, even more complex than the one without negative premises. For the issues in common with the earlier result, the constructions we use have a lot in common, though we do find several simplifications.

This will involve extending our original alphabet Σ_0 in various ways. To avoid confusion and ambiguity we assume that all the various directions we extend it are disjoint from each other and Σ_0. Σ will mean the alphabet in use at any particular time, as extended.

1. First we consider the case of homogeneous combinators (no negative premises), so that the result process is the same format as the original, and where there is no termination. Thus we consider operators whose combinators are all of the form (p, a), with p having no negative aspect and no $\checkmark$ in either p or a. In this case there is no point in considering inactive arguments, since this type of combinator is incapable of activating them.

2. Next we consider how to add similarly restricted n-combinators. Clearly this is the heart of the extension to the original construction.

3. The next step is to allow actions to throw away active arguments. One cannot simply leave the process representing a discarded argument running but blocked in a simulation like this: if it were still running its patience rule would promote its τs, making the simulation a lot less accurate, particularly because it could create a divergence that was not present with the original operator.

4. We then allow non-homogeneous combinators, but only ones that use the existing active arguments rather than inactive ones.

5. The penultimate stage is to show how to use inactive arguments.

6. We finalise the construction by allowing termination.

At each stage the simulation we build takes the form of the parallel composition of processes representing each argument that is active, plus additional parallel components to regulate behaviour, and "zombie" processes representing those that have been inactivated or have terminated. Both the argument processes themselves and the top level are subject to a lot of renaming, plus various other manipulations.

4.1 Homogeneous positive combinators with no termination

In this case (in a simplification from the construction in [27, 26]), the simulation will take the form

$$(((\|_{i=1}^{n}(A_i, P_i[\![R_i]\!])))\ {}_{\cup A}\|_C\ RUN_C)[\![CR]\!] \setminus \{Tau\}$$

where $P_1, \ldots, P_n$ are the (all active) arguments of some operator F. Tau is a member of Σ we introduce to model a combinator generating a τ action. $\cup A$ is the union of the A_i.

Let C be the set of combinators for F. We add C into the alphabet and can construct the renamings as follow

- R_i maps each event a of P_i to each combinator which requires the ith argument to perform a.
- CR maps the combinator (p, a) to a if $a \in \Sigma$, and to Tau if $a = \tau$.
- The alphabet A_i consists of all combinators c which have a proper premise (i.e., not "$\cdot$") in position i.
- The final RUN process is necessary to provide a way in which combinators with no active arguments can happen. It will later be replaced by more elaborate regulator processes.

It should be clear that:

- Any P_i that can perform a τ can perform it in the simulation, with the simulation state progressing exactly as we require in the patience rule that F must have for its ith argument.

- The event representing the combinator c can occur precisely when the premises of c are met (i.e., each non-"$\cdot$" component performs the appropriate event). The renamed P_i can then synchronise to perform c, which CR and the hiding of Tau combine to turn it into the event that c generates. Again the successor state (with just the P_i that contribute to c progressing) is exactly the one that simulates the state that the combinator semantics will have reached under the same action.

- Every state reachable from $F(P_1, \ldots, P_n)$ in our restricted circumstances is of the form $F(P'_1, \ldots, P'_n)$ for P'_i some state of P_i, and thanks to the above observations this state is strongly bisimilar – indeed isomorphic in the sense of transition systems – to the state

  $$((\|_{i=1}^{n}(A_i, P'_i[\![R_i]\!])) \;{}_{\cup A}\|_C\; RUN_C)[\![CR]\!] \setminus \{Tau\}$$

  of the simulation.

  If the result event of a combinator were ✓, we would naturally expect CR to rename that combinator to ✓. However CSP renaming is not permitted to do this: ✓ can neither be renamed nor can anything else be renamed to it. We will see how to solve this problem in Section 4.6.

We have therefore completed the construction, and demonstrated the truth of the theorem, in this first case.

When adding combinators to the alphabet, we will label each with the operator it comes from. This labelling will only be needed in the later parts of this construction.

4.2 Adding negation

Suppose for the moment that no combinator has both positive and negative premises for the same active argument. Then we can get the argument process if necessary to contribute one or other to the firing of the combinator. We know how to achieve this for positive ones. For negative ones we take a lesson from the definition of $\circledcirc_N$ above: we can use priority to deliver an event just when some set of actions is not possible.

For each set of events $S \subseteq \Sigma$ that might (each together with $\{\tau, ✓\}$) be the negative premises for argument process P, let $\neg S$ be a new event that will represent P's inability to perform any of them. Let the set of such $\neg S$ for P (in the context it is placed) be $negs(P)$.[10] Then the process

$$Negate_0(A, P) = \mathbf{Pri}_{\leq_P}(P \;|||\; RUN_{negs(P)})$$

[10]Note that if the alphabet of P is countably infinite, then $negs(P)$ is potentially un-

where[11] $\neg S <_P a$ if and only if $a \in S$, can perform $\neg S$ when P is in a stable state that cannot perform any member of S. This follows the approach taken with events of the form a_1 and a_2 in the definition of $\boxdot_N$. Just as in that case we can check a negative premise on P by getting $Negate_0(P)$ to perform an event as part of a combinator synchronisation whenever that is appropriate. Thus the first component of a combinator now becomes a tuple with components that are either a positive event a, a $\neg S$ or the absence "$\cdot$" of that process's involvement, and the renamings R_i on the component processes of the simulation are extended so the $\neg S$ is renamed to each combinator c that has this as a component at the given process's place.

Note that the effects of priority are local to each argument process: in particular, one process can be in a position to perform a $\neg S$ even though another one is not stable. This is important to preserve the proper simulation of patience rules. The author attempted to find a solution in which a single priority operator was applied at the top level in the simulation, but this failed to work precisely because it sometimes prevented τ actions that were required.

This deals with the situation where no argument has positive and negative premises in the same combinator. But such situations do arise, for example the semantics of the priority operator itself need this: $\mathbf{Pri}_{\leq}(P)$ can only perform non-maximal a if P itself can, but can not perform any higher priority event.

To handle this we introduce yet further events: $(a, \neg S)$ (with $a \notin S$) means that the process can perform a while in a stable state where no member of S can happen.

Whereas we needed the extra RUN process for the simple $\neg S$ events, we do not need it for $(a, \neg S)$, since the correct way to handle this is to make it a renamed copy of a. We therefore extend our previous manipulation of P to

$$Negate(P) = \mathbf{Pri}_{\leq_P}(P[\![NegR]\!] \;|||\; RUN_{Neg(P)})$$

countable! While CSP makes sense over an uncountable alphabet, this is a little embarrassing, though restricting to the set of negative premises that are actually used in the usually countably many combinators for a countable alphabet will then avoid such a blow-up.

[11] In describing this and later partial orders on events, the order being defined will have no more pairs than are implied by transitivity and our postulates about the behaviour of τ and $\checkmark$ in priority orders. So in particular the one described here has exactly the orderings mentioned, under the assumption that the S of $\neg S$ implicitly contains τ and $\checkmark$.

where $NegR$ maps each event a in P's alphabet to both itself and the $(a, \neg S)$ we introduced above, and $\leq_P$ is extended so that $(a, \neg S)$ is given the same priority as $\neg S$. Thus $(a, \neg S)$ can happen just in those stable states where a can be performed by P but no member of S can be.

The extension of the simulation is obvious: the renaming R_i is extended so that each event of the form $(a, \neg S)$ is mapped to every combinator c which has this particular pair of premises for its ith argument.

This clearly maintains our isomorphism.

We have had to do quite a lot to the original simulation to allow for negative premises, not least introduce a lot of extra events. However in any given case simplification is likely to be possible: there is no need for the complications of *Negate* on an argument if there are no negative premises on it, and even then we might get away with $Negate_0$ if no events of the form $(a, \neg S)$ are required. And in general we only need to include $\neg S$ for sets of negative premises that n-combinators actually use.

4.3 Turning processes off

In the rest of this construction there is little difference in how we treat combinators and n-combinators, so from here on we will include both sorts under the term "combinator".

This and the following sections of this construction follow similar methods to those used in [27, 26], with a few modifications arising from the negative premise events and use of combinators directly as events.

With a non-homogeneous combinator, some of the arguments that were previously active may no longer be needed in the result state. Since our simulation had a parallel component for each such argument before the action, it will still have it afterwords, as CSP does not provide a way of reducing the number of components in a parallel composition while some components are still active. Therefore we need to ensure that such components can be turned into zombies that do not affect the operational semantics thereafter. As we said earlier, we need to ensure that the actions we cannot stop the actual argument process from performing (τ and $\checkmark$) do not interfere with the operational semantics once the argument is supposed to have disappeared. So if it is still around we have to make sure it does perform them. What we therefore do is put a harness $Switch(\cdot)$ around the argument which allows it to disappear in favour of something that really behaves like a zombie (which at present means $STOP$, though it will get more complex later). The event that triggers this will be one of the combinators firing. There are separate cases we have to consider for this: either the process which is being shut

down contributes a premise to the combinator or it does not. If it does then the argument (as modified by our previous mechanisms) must communicate the event that closes it down, and otherwise it does not. Let $C1$ be the set of combinators of the first sort (i.e., closing this argument with it communicating) and $C2$ those of the second sort.

The argument process including its harness will have to communicate both of these sorts of combinators, but the process itself only the first. In the first case we shut the process down using the throw operator of CSP: if Z is our zombie process then $P\,\Theta_{C1}\,Z$ allows P to behave normally until it communicates a member of $C1$ and then becomes Z. In the case of $C2$ combinators, P does not contribute to its own disappearance. For $C2$ events we have the choice of using $\triangle$ (interrupt) instead of Θ_A, and indeed this was done in [27, 26]. However, since $\triangle$ can be expressed in terms of Θ_A and other CSP operators [23], it should not be surprising that we can do our whole job with the throw operator. We can define

$$\mathit{Switch}(C1, C2, P) = (P \;|||\; RUN_{C2})\,\Theta_{C1 \cup C2}\,Z$$

It makes no sense to include this feature in our simulation at this stage, since we have not yet introduced the rest of the mechanisms required to handle non-homogeneous combinators. Rather it provides an important part of these mechanisms.

4.4 Non-homogeneous combinators

We will now deal with the case where we are allowed non-homogeneous combinators: ones where the format of the running process can be changed by actions. However we will not yet contemplate situations in which new argument processes (either constants introduced by operators or previously inactive arguments) are set running. Thus the set of argument processes involved will, in the case we are considering now, always be a subset of those active at the start.

Thanks to restrictions placed above we know that the successor syntax T is always either a constant process, a simple argument or a simple operator application satisfying the immediately distributive condition from before.

Any one of these can lead to the switching off of a number of the arguments of the original operator F as detailed above.

The great thing about an operator with only homogeneous combinators is that its arguments are always in the same relationship with each other, and the same set of combinators always apply. However as soon as the structure of the simulation can change, this is no longer true. In this latter case we

need to program each argument (via harnesses and renamings of events) to handle all of the operator contexts it might find itself in. Thus, for example, each event needs to be renamed to all the combinators it participates in in every such context.

However while the arguments have one operator applied, we cannot allow the combinators of other operators to fire. Therefore we need to add a regulator process to our simulation which always knows the current format:

- What operator is being applied...
- to which of the arguments of the original operator.
- It only permits combinators appropriate to this to fire...
- and has its own state changed by the firing of a non-homogeneous combinator.

This will replace the RUN_C component of the simulation to date.

The operation of the regulator can take one of three forms

- After all arguments have been switched off, it might be acting as some constant process.
- After all but one arguments have been switched off, it might be permitting that process to act as itself (i.e., when the result of a previous combinator has been that this single argument is now how the complete system acts). The combinator $((a, \cdot), a, \mathbf{1})$ of $\Box$ is an example.
- It might be allowing the combinators of some operator F, with a mapping from the active arguments of that operator to the running argument processes, with all the others turned off.

In fact the first and second cases can be included in the third, since

- We can safely include all the constant processes that the various combinators might introduce amongst an extended set of inactive arguments. So instead of starting up a constant process (either as the result state of a combinator, or as an argument to the successor combinator), the same constant process can be extracted from the inactive arguments, once we have built mechanisms for doing this.
- It is easy to devise combinators to implement the identity operator on the jth argument: $((\cdots, a, \cdots), a)$ for each a (including $\checkmark$), with the non-"$\cdot$" premise in the jth place. The accuracy of this representation

depends on the rest of the arguments having been turned off, because of patience rules.

We will therefore just deal with the third case above.

In this F is an operator with a (n-)combinator operational semantics and we define

$$\begin{aligned} \mathit{Reg}(F,\xi) \quad = \quad & \Box_{c\in HC(F)}\, \xi(c) \to \mathit{Reg}(F,\xi) \\ & \Box\, \Box_{c\in NHC(F)}\, \xi(c) \to \mathit{Reg}(S(c), \xi.\zeta(c)) \end{aligned}$$

where $HC(F)$ and $NHC(F)$ are respectively the homogeneous and non-homogeneous combinators of the operator F, $S(c)$ is the successor syntax of c, and $\zeta(c)$ is the mapping which says, for each active argument of $S(c)$, which active argument of c it takes over. If c has k arguments and there are n arguments in the initial configuration, ξ is the current (necessarily injective) mapping from the k indices of c to $\{1,\ldots,n\}$ that says where the arguments of c are amongst the n. $\xi(c)$ lifts one of c's combinators to one on the n argument components of the simulation by mapping the positions of the argument processes. (Thus one of the arguments not in the image of ξ will always have the null premise "$\cdot$".) $\xi.\zeta$ means functional composition. Note that you can calculate from ξ and from the expanded combinator $\xi(c)$ which argument processes are no longer in scope and must be zombies.

Note that since the recursion defining $\mathit{Reg}(F,\xi)$ here is transparently guarded, the operational semantics will not insert any extra τ actions from unfolding into this: its only actions are combinators and those directly attributable to any constant process.

Thus, with n active arguments initially, the combinator events performed by the simulation are always based on n arguments, even though the current active operator might have less than n. It is up to us to maintain the invariant that the component processes of the simulation corresponding to arguments no longer in the image of ξ have been switched off as described in the previous section.

The a, $\neg S$ and $(a,\neg S)$ events of each (*Negate*d where necessary[12]) argument process now need to be renamed to all the combinators $\xi(c)$ in which the given argument might participate for all the operators and ξ reachable from the initial configuration.

[12]Where this is necessary if it is in any of the operator contexts that the argument can reach.

The overall simulation now looks like

$$((\|_{i=1}^{n}(A_i, Switch(C1_i, C2_i, Negate(P_i)[\![R_i]\!])) \\ \|_{C}^{\cup A} Reg(F, id))[\![CR]\!] \setminus \{Tau\}$$

where

- A_i remains the set of combinators involving the ith argument, including $C2_i$, now expanded to be n-ary as is done by ξ above.
- $C1_i$ are the combinators that discard/turn off the ith argument in which it participates in the premises.
- $C2_i$ are the combinators that discard/turn off the ith argument in which it does not otherwise participate.
- R_i maps each event of $Negate(P_i)$ to the appropriate combinator(s) in A_i.
- F is the initial operator of the term.
- C are all combinators of all operators involved, expanded to be n-ary by every ξ that might arise. We assume that the combinators from different operators are disjoint, and disjoint from all other events used here.
- id is the identity function on $\{1, \ldots, n\}$, where F has n (active) arguments.

Though by now it is quite complicated, the justifications of the previous steps should make it clear that we still have isomorphic transition systems.

4.5 Making use of inactive arguments

An inactive argument can never have any effect on the transitions of a combinator operational semantics unless it becomes active, necessarily through some non-homogeneous rule activating it. Our definition also allows combinators to activate constant processes, but for brevity we will simply assume that all such constant processes are included in $\mathbf{Q}$, by expanding this and I if necessary. In order to handle activation in our simulation we need:

- A way of managing the arrangement of inactive arguments as the simulation evolves.

- A way of using the combinator that activates an argument to create a copy of the latter that thereafter runs as one of the active arguments. In general an active argument can have several copies started at the same time, and yet remain an inactive one as well.

The effect of starting up inactive arguments can be to *increase* the number of active arguments in a simulation. We have two choices of how to manage this.

- The first is to add new processes onto the end of the row of argument processes each time a new process is started up. In this way the width of the simulation increases each time this happens, and there is no bound to its width. This is achieved by having a single process that can be instructed to construct, dynamically, finite collections of processes to run in parallel with it and the rest of the simulation. This was the approach taken in [27, 26].

- The second is enabled by the limitation we have placed in this paper, because of the effects of negation, on the syntax of the third component of combinators. The most complex thing we allow there is a single operator applied to a selection of argument processes. If we assume that the number of active arguments of all operators that are reachable within the operational semantics of the initial ones is bounded, then we know that even though the number of distinct active arguments that have ever existed in the the simulation is unbounded, there is a bound on the number active at any one time. In this case we can opt to have a fixed number of processes in our simulation, which have the ability to come back from the dead after becoming zombies. Specifically, combinators that activate arguments will command some zombie to reboot itself as any process that might start during the simulation, namely any initially inactive argument (including any constant process that an operator can introduce). In order to avoid the scenario of having to stop an argument and start it again as something else *in the same action*, we will assume there are enough processes to accommodate the stoppings and startings of each combinator being disjoint.

  This ability needs to be programmed into the definition of a zombie from the start, which means this definition now depends on the operators and inactive arguments.

  The advantage of this approach is that there is much more chance of keeping the alphabet as well as the simulation finite.

Neither of these approaches is simple to program in CSP. For details of the first, which would work equally well in our context, see the earlier works. For the second, we need to index the zombie process so that it becomes a suspended process at index position i, and note that if the initial number of active arguments (i.e., those of the initial operator) is less than the number of active arguments we might need later, then we will need to supplement the initial active arguments with some number of zombies so that we have enough.

If Z_i is the process at index position i it must now synchronise on all combinators that have the effect of "reincarnating" it. So for each expanded combinator (analogous to $\xi(c)$ but now taking account of changes in the mapping of inactive arguments as well) that has this effect on some argument we need to calculate which zombies have to come to life as what. We assume the existence of

- $Revive(i)$, the set of expanded combinators that tell a zombie to reactivate.
- For each $c \in Revive(i)$, the index $Reborn(i, c)$ to the original inactive arguments, which is how the component is reborn.

To achieve this despite changes to the operator we will have the regulator process modify the ways in which combinators point at inactive arguments. Specifically, a combinator points at these through indexes in the final (T) component of a non-homogeneous combinator. As constructed for a given operator, the indexes are formed relative to those of that operator. But the zombies were created from processes set up relative to the indexes of the initial operator. Therefore – just as happens for the active arguments – the regulator will modify the indexes in such T to point at the indexes in the original I that correspond to the desired arguments for the operator that is now active. So the regulator must keep a mapping ψ from the indexing set I_G of the presently active operator G to the original one I. This must be a total but, in this instance, not necessarily injective function.

$Reborn(i, c)$ can be calculated from the expanded combinator since the said combinator will generate a finite list of I-indexes it needs to start, and a zombie at i will start as the jth member of this list – modified to take account of its place in the simulation – if it is currently the jth zombie (which it can tell from the combinator).

So the definition of the zombie process is considerably enriched from $STOP$ to

$$Z_i = \Box\{c \rightarrow Switch(C1_i, C2_i, Negate(Q_{Reborn(i,c)})[\![R_i]\!]) \mid c \in Revive(i)\}$$

Note that like $STOP$ this has the crucial property that when in state Z it cannot perform a τ or $\checkmark$, so that it does not introduce any events into our simulation independently when it is in the zombie state.

The regulator process needs to be updated to take part in the revival of zombies. In particular the mapping ξ from the active arguments of the present operator to the indexes of the corresponding argument processes needs, after an action that activates one or more processes, to be augmented with the appropriate indices mapped to the reborn zombies. Clearly the regulator can calculate what this mapping is. The regulator also has to update the inactive argument mapping ψ.

$$\begin{aligned} Reg(F,\xi,\psi) \quad = \quad & \Box_{c\in HC(F)}\, \xi(c) \to Reg(F,\xi,\psi) \\ & \Box\, \Box_{c\in NHC(F)}\, map(\xi,\psi,c) \to \\ & \qquad Reg(S(c),(\xi.\zeta_1(c)) \cup (\rho(\xi,\psi,c).\psi.\zeta_2(c),\psi.\theta(c))) \end{aligned}$$

where

- $map(\xi,\psi,c)$ corresponds to the $\xi(c)$ we used previously: it uses ξ to map the active arguments of the present operator F to the places they occupy amongst the operators, and uses ψ to map the inactive argument indices of F to the original index set I. (Note that homogeneous combinators do not have such indices, so we can still use $\xi(c)$ for $HC(F)$.)

- $\zeta_1(c)$ is that part of the assignment of c's successor term T's active argument list which is to the active arguments of F.

- $\zeta_2(c)$ is the part that maps to the inactive arguments of F, and are thus the subject of re-births amongst the zombies. Note that the inactive arguments of F still need to be mapped to the original $\mathbf{Q}$ via ψ.

- $\rho(\xi,\psi,c)$ is the mapping from the image of $\psi.\zeta_2$ to the indices of the restarted zombies that will act as them:

  $$\rho(\xi,\psi,c)(Reborn(map(\xi,\psi,c),i)) = i$$

  whenever $map(\xi,\psi,c) \in Revives(i)$

- $\theta(c)$ is the function from the inactive arguments of the new operator invocation to those of the previous one. In a combinator of the form $(p, a, G^{\mathbf{Q}'}(...))$, $\theta(c)$ would be derived from the relationship between the indexing in $\mathbf{Q}'$ and that in F's list of inactive arguments.

As an example, consider the following CSP construction which, for simplicity, we define over only an infinite list PS of processes:

$$ListThrow(A, PS) \quad = \quad head(PS) \,\Theta_A\, ListThrow(A, tail(PS))$$

This has the effect of making each communication of any $a \in A$ by any of $\langle P_1, P_2, \ldots \rangle$ pass control to the next. We can give this a combinator operational semantics, where there is a single active argument and a list of inactive ones. The combinators for $LT_A^{\mathbf{PS}}(P)$ (with P now being the head of the list, and **PS** being the tail), are

- (b, b) for $b \notin A$
- $(a, a, LT_A^{tail(\mathbf{PS})}(head(\mathbf{PS})))$ for $a \in A$

To follow our syntax for third components exactly, we need to treat $tail(\mathbf{PS})$ as an indexed structure of pointers into the structure represented by **PS**. If the indexing set of an infinite list is $\{1, 2, \ldots\}$, then $\theta(c)(i) = i + 1$ for all the non-homogeneous combinators c and all i, and after k non-homogeneous combinators we will have $\psi(i) = i + k$. The simulation needs two slots $\{1, 2\}$ where an active process can be running so that we do not have to stop and start the same one each time an A action happens. Each non-homogeneous combinator closes one of these to become a zombie, while re-starting each as the next member of PS that is due to run. So for each such combinator $\zeta_1(c)$ is empty, while $\zeta_2(c)$ maps whichever of 1 and 2 is presently a zombie to 1 (as the pointer to the head of the present inactive argument list), which ψ then maps to the head of the current list of arguments where it sits in the original list, namely the $k + 1th$ where there have been k elements of A so far in the trace.

4.6 Termination

So far, remarkably, we have managed to keep our simulation so exact that the operational semantics of the simulation is isomorphic to that of the target. As we said earlier, and implied in the formulation of Theorems 1 and 2, we have to be a little less exact when the ✓ action is involved. This is because of its nature as a signal and the limitations this places on how the simulation can manipulate it – the reader will have noticed that events in Σ have been manipulated ruthlessly so far in our simulation.

The trick to handling termination within our simulation is to replace each argument process that might terminate by P; $(tick \rightarrow STOP)$ where $tick$ is a new member of Σ distinct from ✓. This new event can be renamed, hidden

and synchronised like any other member of Σ. Wherever a combinator c demands that an argument process communicates $\checkmark$ from argument i, we will rename the *tick* of the said argument to c rather than attempting to illegally rename $\checkmark$ to it. Notice that this combinator will have to synchronise with the regulator process, and possibly others if it turns off more than just the process that is performing the $\checkmark/tick$. The behaviour of the resulting simulation is the same as if the argument P had been replaced by P; *SKIP*, because in each the actual $\checkmark$ of P becomes a τ, and the way the simulation treats the *tick* from this process is the same as when P; *SKIP* performs $\checkmark$ in the combinator semantics.

There is one more subtlety here, which is that when $\checkmark$ is the result event of a combinator then CSP does not allow us to use *CR* to rename that combinator to $\checkmark$. Therefore we rename such combinators to *tick* and place the entire simulation in the context $(Simulation\,\Theta_{tick}\,SKIP) \setminus \{tick\}$. This accounts for the additional ; *SKIP* after the simulation in the theorems.

The use of the event *tick* has another advantage: at various points in this construction we have added components such as *RUN* in parallel or interleaved with the real arguments. If we were trying to make use of termination per se to make the simulation terminate then we would have to make all these terminate too: this is messy and would also introduce extra τs. The use of $\Theta_{\{tick\}}$ avoids this difficulty.

This concludes our construction and therefore our proof of Theorem 2.

5 Generalising and composing

5.1 Composability or not

One of the beautiful features of combinator operational semantics (without negative premises) is the way one can compose the combinators of a tree of operators representing an immediately distributive term in each active argument separately, to obtain a combinator semantics for the whole term, corresponding to the FDR concept of supercombinator. This underpins the whole implementation of FDR, and indeed a later version of FDR3 will exploit this composability to provide direct support for user-defined combinator semantics.

The following example is taken from [26]. The compound operator $((P \underset{X}{\parallel} Q) \underset{Y}{\parallel} R) \setminus (X \cup Y)$ combines three processes together, so the combinators that represent it have three process arguments **1**, **2** and **3**. All of these are active because neither the hiding nor the parallel operator have any inactive

arguments. For simplicity we are considering only the homogeneous case and will assume that the arguments never terminate.

It is not hard to see how we can feed the result event of the combinators for $\mathbf{1} \underset{X}{\parallel} \mathbf{2}$ into the first input of $\cdots \underset{Y}{\parallel} \mathbf{3}$, yielding the following "supercombinators" for $(P \underset{X}{\parallel} Q) \underset{Y}{\parallel} R$, essentially by substituting[13] them for occurrences of their result events in the higher-level operator's premises:

- $((\cdot,\cdot,a),a)[a \notin Y]$ derived solely from $((\cdot,a),a)[a \notin Y]$
- $((a,\cdot,\cdot),a)[a \notin X \cup Y]$ derived by composing $((a,\cdot),a)[a \notin X]$ and $((a,\cdot),a)[a \notin Y]$
- $((\cdot,a,\cdot),a)[a \notin X \cup Y]$ from $((\cdot,a),a)[a \notin X]$ and $((a,\cdot),a)[a \notin Y]$
- $((a,a,\cdot),a)[a \in X \setminus Y]$ from $((a,a),a)[a \in X]$ and $((a,\cdot),a)[a \notin Y]$
- $((a,\cdot,a),a)[a \in Y \setminus X]$ from $((a,\cdot),a)[a \notin X]$ and $((a,a),a)[a \in Y]$
- $((\cdot,a,a),a)[a \in Y \setminus X]$ from $((\cdot,a),a)[a \notin X]$ and $((a,a),a)[a \in Y]$
- $((a,a,a),a)[a \in Y \cap X]$ from $((a,a),a)[a \in X]$ and $((a,a),a)[a \in Y]$.

In other words – exactly as we would expect – there is one rule for each way in which events of the three arguments might synchronise doubly or triply or happen independently.

The effect of the hiding operator is to take those combinators where the action a is in $X \cup Y$ and turn it into τ.

This composability of combinators underlies the liberal syntax we adopted for the final components T of combinators, as opposed to n-combinators. For we can treat any such term as an operator with naturally constructed combinator operational semantics in its own right.

If the same were true of n-combinators, then there would be every chance of extending FDR straightforwardly and efficiently to encompass n-combinator definable operators. Regrettably this is not true. As a simple example consider the operator

$$\mathbf{Pri}_{\leq_1}(\mathbf{Pri}_{\leq_2}(RUN_{\{b,c\}} \;|||\; P))$$

in which $c <_1 b$ and $b <_2 a$. On a stable state of P, this process can perform c whenever P can perform a from a stable state, but the combinator (a, c) is

[13]The exception to this is where the result action of the lower-level combinator is τ. However if we were actually to instantiate a patience rule as $((\cdots,\tau,\cdots),\tau)$ and substitute into that, we would get the correct result.

not an accurate representation of this behaviour, since in the operator above the c occurs without P progressing, whereas in the combinator semantics P would do the a and move to its next state.

It is interesting to note that applying a single priority operator with the order $c < b < a$ to $RUN_{\{b,c\}} \mid\mid\mid P$ would not achieve this effect: in fact c would never be possible as the process inside the priority operator can always perform b.

This impossibility of composing n-combinators and getting another one as we could with combinators accounts for the more restricted form of T we use in this paper, because in the CSP-like case we could depend on a complex term T having its own (super)combinators, meaning that we can think of it as a CSP-like operator in its own right.

5.2 Probing

The example above strongly suggests that we could liberalise the condition on n-combinators that an argument process must progress if the premise is that it can perform some positive action, via something akin to double negation. This is a form of *probing* a process as discussed in [2], which in general allows an operator to inspect the complete acceptance/ready set of its argument(s) before deciding what to do next. The idea would be to replace each argument with a version in which first each alphabet member was renamed to two copies of itself, with one copy being used to make progress and the other to probe availability without moving the process forward. By elaborating on the construction above we could create a context for P in which we had events

- a, representing the actual occurrence of the event.
- $probe.a$, representing the availability of a but not executing it.
- $\neg S$ for each set of events S

This can be achieved by the construct

$$(\mathbf{Pri}_{\leq_1}(RUN_{\{probe.a \mid a \in \alpha P\}} \mid\mid\mid \mathbf{Pri}_{\leq_2}(RUN_{\{a' \mid a \in \alpha P\} \cup \{\neg S \mid S \in negs(P)\}} \mid\mid\mid P))) \underset{\{a' \mid a \in \alpha P\}}{\parallel} STOP$$

where $\leq_1$ makes $probe.a$ less than a' and $\leq_2$ makes a' less than a and $\neg S < a$ if $a \in S$.

This can in fact be extended to probing for the availability of a set of actions: let $probe.B$, for $B \subseteq \Sigma_0$, mean establishing that all of B is available from this state. Then we can adapt the above to achieve this effect via

$$(\mathbf{Pri}_{\leq_1}(RUN_{\{probe.B|B\subseteq\alpha P\}} \;|||\; \mathbf{Pri}_{\leq_2}(RUN_{\{a'|a\in\alpha P\}\cup\{\neg S|S\in negs(P)\}} \;|||\; P))) \underset{\{a'|a\in\alpha P\}}{\|} STOP$$

where now $probe.B \leq_1 a'$ just when $a \in B$.

Furthermore we can actually perform a positive and negative probe simultaneously if we extend *probe* to have both a positive set B and a negative one $\neg S$:

$$(\mathbf{Pri}_{\leq_1}(\mathbf{Pri}_{\leq_2} RUN_{\{a'|a\in\alpha P\}\cup\{\neg S|S\in negs(P)\}} \;|||\; P) \; [\![probe.B.\neg S / \neg S \mid S \in negs(P), B \subseteq \alpha P]\!])) \underset{\{a'|a\in\alpha P\}}{\|} STOP$$

where again $probe.B.\neg S \leq_1 a'$ just when $a \in B$.

Note that

- Both positive and negative probing of P require that P is in a stable state.
- Plainly $\neg S$ can happen in the previous version if and only if $probe.\emptyset.\neg S$ can happen in this one, since the latter event is maximal under $\leq_1$.
- $probe.B.\neg S$ can thus happen whenever no member of S can occur in P and no a' for $a \in B$ can occur. The latter means that every $a \in B$ can occur in the same state of P and the same one where the $\neg S$ is true.

We could even add a single ordinary positive premise into this mix, namely an event that P performs as part of the rule by adding the usual events $(a, \neg S)$ as additional renamed copies of the a from P, giving it the usual priority below members of S in the inner priority operator. Outside that priority that event is renamed to $ProbedAction.a.B.\neg S$ for every B alongside $\neg S$ being renamed to $probe.B.\neg S$ and putting the $ProbedAction.a.B.\neg S$ below a' for $a \in B$ in the outer priority operator.

Given these observations, it would have been reasonable to have extended the definition of Pri-CSP-like operational semantics to allow a positive probing premise (meaning that a set of actions must all be available and the state is stable) on each active argument in addition to those we have used already. The reason we did not do so were:

- The additional levels of complexity it adds to the simulation.
- The fact that, since $\mathbf{Pri}_{\leq}(\cdot)$ is itself expressible in the Pri-CSP-like semantics we have defined, so are all the above constructions in multiple stages.
- It seems unlikely that pn-combinators as defined below will have good composability properties. We will discuss this below.

However we may define

Definition 3 *A pn-combinator is a combinator in which each active argument may have any or all of a normal positive premise, a negative premise, and a* probing *premise of a set of events. (So the p of pn here stands for probe.)*

Definition 4 *An operator has* Generalised *Pri-CSP-like operational semantics if it can be defined in pn-combinators.*

It is clear how we could have generalised our earlier construction to incorporate the above, so we have the following theorem.

Theorem 3 *Suppose the operator $F^{\mathbf{Q}}(\mathbf{P})$ has generalised Pri-CSP-like operational semantics, together with all other operators reachable (transitively) through the T third components of its combinators. Then for any arguments $\mathbf{P}$ and $\mathbf{Q}$, $F^{\mathbf{Q}}(\mathbf{P})$ is expressible in Pri-CSP in the sense that the simulation is strongly bisimilar to $F^{\mathbf{Q}'}(\mathbf{P}')$; SKIP where $\mathbf{P}'$ and $\mathbf{Q}'$ are defined as previously by replacing each component P by P; SKIP.*

The problem with composability for pn-combinators is that the only piece of information a single such combinator can deliver us about a process is a single action that an operator can perform when applied to some arguments. In order to compose pn-combinators in a direct way the combinator would have to tell us the current acceptance/ready set of the composition in any stable state. While we could presumably add this information into the combinators, they would look much less like an operational semantics and more like ready-trace/$\mathcal{FL}$ semantics.

6 Comparisons

Many of the concepts discussed in this paper – though not relating to negative premises, probing or priority – are rooted in the author's own 1982

doctoral thesis [20], where a section was devoted to an early derivation of some of the "CSP-like" qualities of CSP's operational semantics.

As stated earlier, the closest comparison with our work on CSP-like operational semantics that we are aware of is with van Glabbeek's [11][14] concept of simply WB-cool operational semantics. This is more liberal than CSP-like because it permits cloning and because it allows arbitrary probing of active arguments in the sense described above: you are allowed multiple premises of the form $P \xrightarrow{a} P'_a$ for different as, and we can choose to use either the results P'_a or the original P in the result term. (Note that more than one can be used because of cloning). Van Glabbeek also allows active arguments to become inactive under limited circumstances. The restrictions there are all expressed in the language of SOS. Van Glabbeek shows that such semantics ensure congruence under weak bisimulation.

[11] also introduces some variants on WB-cool which have congruence properties for different forms of bisimulation.

Both CSP-like and Pri-CSP-like operational semantics (like strictly WB cool and similar classes) come firmly within the GSOS class of operational semantics defined in [3]. This is well studied, and implies, for example, that [4] the use of negative premises causes no problems with the well-definedness of operational semantics.

The paper [2] provides a very detailed survey of restrictions on SOS semantics which are intended to preserve various forms of congruence. In particular it identifies full probing – namely the ability to test the complete acceptance/ready set as a condition for actions – with the natural notion of operational semantics which coincides with the $\mathcal{FL}$ style of model, there termed ready-trace. It is clear that our notion of Generalised Pri-CSP-like semantics and the proof that all such operators are implementable in Pri-CSP is just another angle on this correspondence.

The author believes, however, that the style presented via combinators, n-combinators and pn-combinators in this paper of giving a much reduced format of simplified transition rules that are guaranteed to be legitimate has much to offer in terms of clarity relative to placing highly technical restrictions on SOS rules.

It is also true to say, of course, that the concise elegance of ordinary combinators is partly lost in their more elaborate counterparts.

In any case the core objective of the present paper is not to capture what style of operational semantics has a given congruence or congruences, but rather to capture a broad range of operational definitions that can be

[14] Van Glabbeek's work was itself closely related to work by Bloom and others [1].

simulated with a high degree of precision in CSP or Pri-CSP. The congruence results we do get are a corollary to this latter objective.

7 Conclusions

One of the most interesting features of this work is the great expressive power of $\mathbf{Pri}_{\leq}(\cdot)$ in conjunction with ordinary CSP. The wide range of effects we have obtained from it in the present paper demonstrate this, as well as our main result and its later extension in Theorem 3.

Whereas the result about CSP-like semantics from [27, 26] had considerable practical consequences for the construction of FDR, this seems unlikely to be true for that about Pri-CSP-like semantics, primarily because of the non-composability of n-combinators. However we should have given those coding in Pri-CSP a good idea of how to achieve some of the less obvious consequences of Theorem 2, including the implementability of $\circledcirc_N$ and the ideas behind $Negate(\cdot)$. We also know a large class of extra operators that, if implemented independently of supercombinators, would make sense in checks over the $\mathcal{FL}$ class of models if these are added to the tool.

An interesting property of $\mathbf{Pri}_{\leq}(P)$, which our constructions make considerable use of, is a consequence of it mapping regular LTSs to regular LTSs. That is, that when P can perform a and b, and $a < b$, then processes like $\mathbf{Pri}_{\leq}(P) \underset{\{b\}}{\parallel} STOP$ deadlock even though some might expect that P being blocked from performing b would re-enable it to perform a. There is no way in which an ordinary LTS representation of $\mathbf{Pri}_{\leq}(P)$ could reflect this nuance.

Carrying on this discussion, let us imagine a model of stable state in which if we try an event we can either see it happen or see it refused. (This is a model that intuitively appeals to the author.) The operational semantics of $\mathbf{Pri}_{\leq}(P)$ appear quite reasonable in this regard: whenever an event is blocked, there is one of higher priority that can happen. We can imagine an implementation trying events one after another from highest priority down until one occurs. This seems completely in line with our imagined model. However it is clear that the discussion in the previous paragraphs invalidates this thinking. An event of higher property that is tried and succeeds, but is externally blocked, would lead the implementation into an inconsistent state (at least if we exclude cloning). It means that to have our priority operator in the context of CSP, we must accept a model in which we can check whether a stable state can perform an event in Σ without giving it the chance to happen. That helps to explain why we have managed to express

probing for the possibility of an event using the priority operator.

The results of this paper also emphasise the huge expressive power of CSP itself, given what we have been able to achieve by the addition of a single operator.

It is reasonable to ask how crucial the choice of priority is for this extra operator. Clearly such an operator cannot be CSP-like, and must have the property that $\mathbf{Pri}_{\leq}(\cdot)$ is expressible using it and the rest of CSP. Not all operators requiring negative premises would do: consider angelic choice $\boxdot_N$ as defined earlier. It seems intuitively obvious that priority cannot be defined in terms of $\boxdot_N$ and the rest of CSP, and to prove this we observe that there are a number of models where $\boxdot_N$ has a compositional semantics but $\mathbf{Pri}_{\leq}(\cdot)$ does not, for example[15] the failures-divergences model $\mathcal{N}$. If $\mathbf{Pri}_{\leq}(\cdot)$ were so definable, it too would have such a compositional semantics, but it does not. Of course the same argument applies to any other operator for which some CSP model is a congruence, that does not work similarly for the general case of $\mathbf{Pri}_{\leq}(\cdot)$. So in particular the restricted cases of priority that are compositional over refusal testing models are also insufficient.

The author has constructed several files to illustrate the ideas set out in this paper, including the implementation of angelic choice using priority set out in Section 3 and the possibility of probing and detecting exact ready/acceptance sets. These are available to download (for running on FDR3) from the author's web site[16].

Acknowledgements

This work was done under funding from the DARPA HACMS programme. It has benefited hugely from discussions with Rob van Glabbeek and Tom Gibson-Robinson.

References

[1] B. Bloom, *Structural operational semantics for weak bisimulations*, TCS, Volume 146, Issues 1-2, July 1995, pp 25-68.

[15] The divergence traces of $P \boxdot_N Q$ are the union of those of P and Q, and aside from the failures implied by divergence strictness, it has the failures (s, X) that belong to *both* P and Q and to just one of them provided there is a prefix $s\langle a\rangle$ of t such that $(s, \{a\})$ is a failure of the other of P and Q.

[16] `http://www.cs.ox.ac.uk/files/6742/pricsplike.tar`

[2] B. Bloom, W. Fokkink and R. van Glabbeek. *Precongruence formats for decorated trace semantics*, ACM Transactions on Computational Logic, Volume 5 Issue 1, January 2004.

[3] B. Bloom, S. Istrail, and A. Meyer. *Bisimulation can't be traced.* JACM **42**(1), 1995.

[4] R. Bol and J.F. Groote. The meaning of negative premises in transition system specifications. JACM, **43**(5), 863-914, 1996.

[5] S.D. Brookes, *A model for Communicating Sequential Processes*, Oxford University DPhil thesis, 1983.

[6] S.D. Brookes, C.A.R. Hoare and A.W. Roscoe, *A theory of communicating Sequential Processes*, Appeared as monograph PRG-16, 1981 `http://web.comlab.ox.ac.uk/people/Bill.Roscoe/publications/1.pdf` and extended in JACM **31** pp 560-599, 1984.

[7] S.D. Brookes, A.W. Roscoe and D.J. Walker, *An operational semantics for CSP*, Oxford University Technical Report, 1986.

[8] C.J. Fidge, *A formal definition of priority in CSP*, ACM Transactions on Programming Languages and Systems, **15**, 4, 1993.

[9] T. Gibson-Robinson, TYGER: a tool for automatically simulating CSP-like languages in CSP, Oxford University dissertation, 2010.

[10] T. Gibson-Robinson, P. Armstrong, A.Boulgakov and A.W. Roscoe. FDR3 – A modern refinement checker for CSP. In *Tools and Algorithms for the Construction and Analysis of Systems* (pp. 187-201). Springer 2014.

[11] R.J van Glabbeek, On cool congruence formats for weak bisimulations. In *Theoretical Aspects of ComputingICTAC 2005* (pp. 318-333). Springer 2005.

[12] M.H. Goldsmith, A.W. Roscoe, P. Armstrong, D. Jackson, P. Gardiner, B. Scattergood and others, *The FDR manual*, Formal Systems and Oxford University Computing Laboratory 1991-2011.

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

[14] A.E. Lawrence, *CSPP and event priority*, Communicating Process Architectures, **59**, 2001.

[15] G. Lowe, *Probabilistic and prioritised models of Timed CSP*, Theoretical Computer Science, **138**, 2, 1995.

[16] R. Milner, *A Calculus of Communicating Systems,* LNCS 92, 1980

[17] R Milner, J Parrow and D Walker, *A calculus of mobile systems, Parts I/II*, Information and Computation, 1992

[18] I. Phillips, *Refusal testing*, Theoretical Computer Science, **50**, 3, 1987.

[19] Plotkin, Gordon D. *A structural approach to operational semantics.* 1981.

[20] A.W. Roscoe, *A mathematical theory of Communicating Sequential Processes*, Oxford University DPhil thesis, 1982.
`http://www.cs.ox.ac.uk/people/bill.roscoe/publications/2.pdf`

[21] A.W. Roscoe, *model-checking CSP*, in *A Classical Mind, essays in honour of C.A.R. Hoare*, Prentice-Hall 1994.

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

[23] A.W. Roscoe, *The three platonic models of divergence-strict CSP* Proceedings of ICTAC 2008.

[24] A.W. Roscoe, *Revivals, stuckness and the hierarchy of CSP models*, JLaP **78**, 3, pp163-190, 2009.

[25] A.W. Roscoe, *CSP is expressive enough for* π, Reflections on the work of C.A.R. Hoare, Springer 2010.

[26] A.W. Roscoe, *Understanding concurrent systems*, Springer 2010.

[27] A.W. Roscoe, *On the expressiveness of CSP*,
`https://www.cs.ox.ac.uk/files/1383/expressive.pdf`, 2011.

[28] A.W. Roscoe and P.J. Hopcroft, Slow abstraction through priority, *Theories of Programming and Formal Methods*, Springer LNCS, 2013.

[29] D. Sangiorgi and D. Walker *The* π*-calculus: A theory of mobile processes*, CUP, 2001.