The expressiveness of CSP with priority

A.W. Roscoe [1]

Oxford University Department of Computer Science

Abstract

The author previously [16,15] defined *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 [15], with the restriction on negative premises removed.

Key words: CSP, operational semantics, priority

1 Introduction

As well as its denotational semantics in models such as traces $\mathcal{T}$ and failures-divergences $\mathcal{N}$, CSP [11] has a well-established operational semantics first described in SOS in [5,6], and congruence with that is perhaps the main criterion for the acceptability of any new semantic model.

The author previously created a class of *CSP-like* operational semantic definitions that automatically have semantics over every CSP model. 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 some action can fire only if one of its arguments *can not* perform some (either one or more) action(s).

There have been a number of proposals for adding priority to CSP. A straightforward one, because it does not involve building special semantic models or types of LTSs, was proposed in [15]. $\mathbf{Pri}_{\leq}(P)$, for a partial order on the events that processes perform, permits P an event x only when no higher priority event is possible. With restrictions on how the invisible event τ fits into $\leq$, this adds very usefully to CSP, for example by permitting the accurate description of real-time systems.

[1] Email: bill.roscoe@cs.ox.ac.uk

This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

$\mathbf{Pri}_{\leq}(\cdot)$ is not CSP-like since it requires negative premises. Indeed it does not have a semantics in most CSP models. This raises the question of whether we can capture a notion of *Pri-CSP-like* operational semantics which includes this operator, where all Pri-CSP-like operators can be expressed in terms of CSP plus $\mathbf{Pri}_{\leq}(\cdot)$. Establishing such a notion is the job of the present paper.

In the next section, we remind ourselves about CSP and its operational semantics. We then recall CSP-like operational semantics and outline their expressiveness result. Finally we recall the definitions of $\mathbf{Pri}_{\leq}(\cdot)$ in terms of operational semantics and over $\mathcal{FL}$, the *finite linear* or *ready traces* model that can record an acceptance set before each event. In Section 3 we generalise the definition of *CSP-like* to achieve the goal set out above. 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 augmented CSP. The precision obtained by this simulation depends on whether or not the language involves the CSP concept of termination, represented $\checkmark$. However, for brevity this paper does not include the role of $\checkmark$ in CSP semantics: it is fully covered in the extended version [18].

As with [16], the primary motivation of this paper is to characterise what operators and languages can be translated into CSP (in this paper extended by $\mathbf{Pri}_{\leq}(\cdot)$) to identify which of these can be handled on the model checker FDR [9], which itself now supports this operator [2]. We give some examples of what is now representable in Section 5.

2 Background

2.1 The operational semantics of CSP

The SOS operational semantics [5,6] of CSP came along after its well-known denotational semantics. For CSP (without $\checkmark$ and sequential composition), the action labels come from $\Sigma \cup \{\tau\}$, where Σ is the *alphabet*, the actions that are visible to and controllable by the external observer, and τ is an invisible and uncontrollable event such that whenever it is enabled and another event does not happen quickly, it will. Given the process P, αP means its own set of Σ actions, which is usually just the visible events it uses.

In SOS style [13] we need rules to infer every action that each process can perform. The conditions that enable actions can be 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.

[2] FDR3 supports two priority operators: `prioritisepo` is directly equivalent to the one used in this paper, while `prioritise` is a restricted case that does not require the programmer to construct an explicit partial order.

- *Negative:* The same except the other process cannot perform a given action.
- *Side conditions* on the actions etc that appear.

A rule has a set of actions/alphabets etc parameters, and some positive and/or negative *premises*. A rule with free parameters other than processes is a rule *schema* denoting a separate formal rule for each permitted value of these.

CSP has a few constant processes, a number of operators which can be applied to argument processes, and recursive constructions. The operational semantics of constants simply describe their actions directly. $STOP$, which has no actions, has no operational rules. Other constants 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,[3] and **div**, which simply diverges: performs an infinite series of τs.

There are two approaches to the operational semantics of recursion:

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

where p is a process identifier and P a process term where 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 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 negative premises.

Without such an undefined recursion (one where the first-step actions 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 $P_i = P_{i+1} \,\Box\, a \rightarrow STOP$), such problems do not arise and (B) gives a more efficient LTS (i.e. less states and transtions). 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.

[3] A formulation of $Chaos_A$ valid in all CSP models has τ transitions to $?x : B \rightarrow Chaos_A$ for every $B \subseteq A$. This can be simplified when only the most common semantic models are in use to have only two states: one which can do a τ to $STOP$ (the other state) or any member of A to itself.

2.2 The transition rules of CSP operators

Communications are introduced via *prefixing* $e \to P$. It has rule

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

Here e may represent a range of possible communications and bind one or more identifiers in P, as in the examples $?x : A \to P$, $c?x?y \to P$ and $c?x!e \to P$. We assume the existence of functions

- $comms(e)$ is the set of communications described by e. For example, $d.3$ represents $\{d.3\}$ and $c?x{:}A?y$ represents $\{c.a.b \mid a.b \in type(c), a \in A\}$.
- If $a \in comms(e)$, $subs(a, e, P)$ substitutes part of a for each identifier bound by e. So $subs(c.1.2, c?x?y, d!x \to P(x, y)) = d!1 \to P(1, 2)$.

Nondeterministic choice picks an argument to act like:

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

The initial actions of prefixing and $P \sqcap Q$ do not depend on those of process arguments. 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. Operators may have some arguments 'active' and some 'inactive'. The former are those whose actions are immediately relevant, the latter the ones which are not needed to deduce the first actions of the combination.

Both arguments of *external choice* ($\Box$) are active. When an argument is active, it must be allowed to perform any τ action it is capable of, since the argument's environment (in this case the operator) is incapable of stopping them. Such τ actions are invisible to the operator, so there are always rules like the following for active arguments:

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

which simply allow the τ to happen without otherwise affecting the process state. These *promote* the τ actions of the arguments to τ actions of the whole process. $\Box$ can use visible actions, here resolving the choice.

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

It is important that the argument P of $e \to P$ is inactive. If not, it would be allowed to perform τs so $a \to P$ might diverge without performing a.

The rules for hiding and renaming both allow all the actions of the underlying process but change some of the names of the events. Renaming applies

a relation to visible ones; hiding turns selected actions into τs.

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

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

We give the semantics of just one parallel operator. Others can be deduced from it: $P \underset{X}{\parallel} Q$ synchronises P and Q on all actions in X, and lets them communicate freely on other events. 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 visible events: two symmetric ones for $a \notin X$

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

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

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

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 A, and Q in B. 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 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 performs $a \in \Sigma$, then the result is interruptable, whereas if Q performs $a \in \Sigma$, then it takes over.

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

The other operator allows any event from P in the set A to close it down and hand over to Q: the *throw* operator $P \Theta_A Q$. P is active: it is allowed to perform τ or $a \notin A$ and carry on whereas $a \in A$ hands control to Q:

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

The operational semantics of $\mathbf{Pri}_{\leq}(\cdot)$ can be found in Section 2.4.

2.3 CSP-like operational semantics

All premises above are positive. The rules also have properties

- If an argument process performs an action $P \xrightarrow{x} P'$ in the premises, and remains after the derived action then P has become P' in the result.
- If an argument process does not act in the premises, then if it remains after the action it stays in its initial state.
- If 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 $F(P, \ldots) \xrightarrow{\tau} F(P', \ldots)$. There are no other rules with τ as a premise.
- No argument process ever appears more than once in the result of any actions. This is the *no cloning* property. In fact CSP can clone inactive arguments via recursion, but never active ones.

In [16,15], 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, so constrained that it can only express CSP-like operators.

CSP-like operational semantics bears close comparison with *simply WB cool* rules as defined in [10]. This is a restriction on SOS that ensures that operators respect weak bisimulation (hence WB). We adopt some of the nomenclature of [10], though this is different from that in [15,16]. This includes the terms *active*, and *inactive* otherwise. [15,16] termed these **on** and **off** respectively. The rules which simply promote a τ action are called *patience* rules.

In giving a combinator 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 F's 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 immediately relevant. We allow an infinite number of components to $\mathbf{Q}$. 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). We only allow finitely many active arguments: not only does the infinite case not arise in CSP, but it is theoretically problematic.

As with SOS, a combinator operational semantics consists 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 $\Sigma \cup \{\cdot\}$. 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 put quotes 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 $^-$. Note that τ is not permitted in these tuples: we will discuss this below.
- The second component is an action y in $\Sigma \cup \{\tau\}$ which represents the result action of the rule: the one that the operator performs when the active arguments perform the components of the first. Hiding gives a case where a visible action is turned into τ, hence the possibility of y being τ.
- The third component represents the successor process after the action. There are two possibilities here:

(i) 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 of parallel, hiding and renaming operators, and combinations of these. The third component is then omitted, so the combinator becomes a pair. Such combinators are *homogeneous*.

(ii) 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 place-holders 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 wishes.

The way combinators 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, common to all non-recursive CSP operators. This is a piece of syntax T 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 $x_i \in \Sigma$.

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

  The pieces of syntax T above can contain arbitrary *closed* CSP processes at any point without restriction

Hiding $P \setminus X$ has rules $(a, a)[a \notin X]$ and $(a, \tau)[a \in X]$, using the convention that for operators with a single active argument, we write a rather than (a) for the first component. The result of $P \setminus X$ processing an action $P \xrightarrow{a} P'$ is always $P' \setminus X$, so its combinators are homogeneous. On the other hand, the resolution of $P \,\Box\, Q$ does change the process structure, so its rules are $((a, \cdot), a, \mathbf{1})$ and $((\cdot, a), a, \mathbf{2})$: either side can perform any action in Σ, resolving the choice. We do not write down patience rules since they *always* apply.

Definition 2.1

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

Theorem 2.2 *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 $F^{\mathbf{Q}}(\mathbf{P})$ and $F^{\mathbf{Q}}_{CSP}(\mathbf{P})$ are strongly bisimilar.*

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

The proof can be found in [16,15] [4]. and is indicated in that of the main hteorem of this paper.

2.4 Priority

While there have been a number of versions of CSP with priority, for example [12,7], the one we use in this paper is that introduced in [15]. This is conceptually simple, as it does not require any re-interpretation of LTS's or CSP models as entities where one action has priority over another. Instead $\mathbf{Pri}_{\leq}(\cdot)$ inputs an ordinary LTS and the result is another ordinary one. $\leq$ is

[4] Tom Gibson-Robinson [8] implemented the constructions of [16], thereby providing a translation of arbitrary CSP-like operators into CSP for use on FDR [9].

a partial order on events $\Sigma \cup \{\tau\}$ which is subject to several conditions stated below. The SOS operational semantics are

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

P performs actions that are not strictly lower under $\leq$ than another action that P can perform from the same state. In the above, x and y range over the whole of $\Sigma \cup \{\tau\}$. In order to make this consistent with the tenets of CSP we need to respect the idea that τ is not controllable and that every process is equivalent to the one where a single τ precedes it:

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

Only the richest CSP models make $\mathbf{Pri}_{\leq}(\cdot)$ compositional. Of those discussed in Chapters 10, 11 and 12 of [15], the only ones compositional for the full range of permitted $\leq$ are the $\mathcal{FL}$ class of models, recording traces extended by one of the following before each event and after the last:

- $\bullet$ meaning that the state from which the next event happened, or at the end of the behaviour, has not been observed to be stable (i.e., a state where no τ is possible).
- Where stability is 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 semantics of $\mathbf{Pri}_{\leq}(P)$ over $\mathcal{FL}$ are as follows, quoted from [17].

$$\{\langle A_0, b_1, A_1, \ldots, A_{n-1}, b_n, A_n\rangle \mid \langle Z_0, b_1, Z_1, \ldots, Z_{n-1}, b_n, Z_n\rangle \in P\}$$

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 $A_i = \{a \in Z_i \mid \neg\, \exists\, b \in Z_i.b > a\}$.
- In each case where $A_{i-1} \neq \bullet$, $b_i \in A_{i-1}$.

Priority is not CSP-like, so we name the extended language *Pri-CSP*.

3 What can we express in Pri-CSP?

$\mathbf{Pri}_{\leq}(\cdot)$ has some of the qualities of CSP-like operators, for example it has the patience property, and never clones its argument. The only one it obviously fails is the ban on negative premises.

To grasp what can be expressed in Pri-CSP we change the expressive power of combinators. Recall that the first component of a combinator 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 is a set of events, which if non-empty contains τ (if not written down it is assumed implicitly), that the process must not be able to perform if the rule is to fire. We annotate such negative premises with the negation symbol $\neg$. A negative premise can only be satisfied in a stable state of its argument.

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 $\{\tau\}$)) we will just write the other, and if both are trivial we will just write "$\cdot$".

There is no difference in the second component of combinators. However, problems discussed fully in [18] make us more restrictive in the syntax of the allowed third component syntax T. Specifically we restrict the third component to be any of

- One of the argument processes by itself (a common case in CSP): this can be active or inactive 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.

We again assume a patience rule for each active argument. A *homogeneous* n-combinator is one where the third component is omitted because the result has the same structure as the initial process.

Any combinator with a negative premise is 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 these extended combinators. It has 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 $((a, \neg\{x \in \Sigma \cup \{\tau\} \mid a < x\}), a)$, where we note that the set of negative premises always includes τ due to the restrictions placed on $\leq$ in the definition of the priority operator.

Recall that the operator $P \,\Theta_A\, Q$ 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 $P \Delta\Theta Q$ 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 $a \in \alpha P$ 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, we will show how to implement

this.

Definition 3.1 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.

4 Expressibility theorem

Theorem 4.1 *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 $F^{\mathbf{Q}}(\mathbf{P})$.*

This implies that such operators have a compositional semantics over $\mathcal{FL}$.

As in [16,15], our proof is to construct the (Pri-)CSP implementation. This is 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.

(i) First we consider the case of homogeneous combinators (no negative premises). Thus we consider operators whose combinators are all of the form (p, a), with p having no negative aspect.

(ii) Next we consider how to add similarly restricted n-combinators. This is the heart of the extension to the original construction.

(iii) The next step is to allow actions to throw away active arguments.

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

(v) The final stage is to show how to use inactive arguments.

In this version of the paper we concentrate on the first two stages. The rest are given in detail in the extended version. 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.

4.1 Homogeneous positive combinators

In this case (in a simplification from the construction in [16,15]), the simulation will take the form

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

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 follows.

- 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 $a = \tau$.
- A_i consists of all combinators c which have a proper premise (i.e., not "$\cdot$") in position i.
- The RUN process provides a way in which combinators with no active arguments can happen. It is later replaced by more elaborate regulator processes.
- 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 into its own second component. 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 by the above observations this state is strongly bisimilar – indeed isomorphic in the sense of transition systems – to the following simulation state.

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

We have therefore completed the construction in this first case.

4.2 Adding negation

Suppose for the moment that no combinator has both positive and negative premises for the same 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 premises we can use priority to deliver an event just when some set of actions is not possible.

For the $S \subseteq \Sigma$ that might (each together with $\{\tau\}$) be 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 $negs(P)$. Then $Negate_0(A, P) = \mathbf{Pri}_{\leq_P}(P \,|||\, RUN_{negs(P)})$ where $\neg S <_P a$ 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. We can check a negative premise on P by getting $Negate_0(P)$ to perform an event as part of a combinator synchro-

nisation whenever that is appropriate. 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. The renamings R_i on the components are extended so the $\neg S$ is renamed to each combinator c that has $\neg S$ as a component at the given process's place.

There are cases with positive and negative premises on the same argument, such as the semantics of the priority operator itself: $\mathbf{Pri}_{\leq}(P)$ can only perform non-maximal a if P itself can, but cannot perform any higher priority event. To handle this we use 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. The above definition is extended to $\mathit{Negate}(P) = \mathbf{Pri}_{\leq_P}(P[\![\mathit{NegR}]\!] \,|||\, \mathit{RUN}_{\mathit{Neg}(P)})$ 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. To handle this the renaming R_i is extended so that $(a, \neg S)$ is mapped to every combinator c which has this particular pair of premises for its ith argument.

4.3 Further stages

The rest of this proof follows similar lines to the one in [16] without priority. This is set out in detail in the extended paper [18]. To handle processes being discarded (as can happen to either argument of $P \,\Box\, Q$ and the first arguments of $P \,\triangle\, Q$ and $P\,\Theta_A\,Q$) we place each $\mathit{Negate}(P_i)$ in a harness, where P_i can be switched off by Θ_A whether or not P_i itself participates in the (necessarily non-homogeneous) combinator that causes this effect. A strong sense of how this is done is given by the deadlock-exception catching operator $P \Delta\Theta Q$ we described earlier: it can be written $((\mathbf{Pri}_{\leq}(P \,|||\, \delta \to \mathit{STOP}))\,\Theta_\delta\, Q) \setminus \{\delta\}$ for δ a new event, the only ordering by $\leq$ being to place δ below all others.

The other aspects of non-homogeneous combinators that need to be handled are (i) allowing the sorts of continuation permitted by the third component syntax T and (ii) activating inactive arguments to participate in the operations of such T. The first is achieved by extending the alphabet to include labelled (n-)combinators for every form that the system might evolve to as the simulation progresses, together with a regulator process which understands what the present format of the system is and how the current active argument processes map onto the format's active arguments.

There are two ways of handling the activation of inactive arguments: one each is described in [16] and [18]. The first dynamically generates new argument processes each time one is activated. The second is possible where the overall number of active arguments has an upper bound, and works by recycling them: letting the zombies created by turning arguments off be reborn in a new form.

All of this can be be done in such a way as to obtain strong bisimulation.

5 Examples

We have seen how to create an operator that allows deadlock in one process to cause a second process to start. The following transformation provides the basis for many similar constructions.

Suppose P has alphabet Σ_0, and that we have added as follows to the overall alphabet: $\Sigma_1 = \{\neg a \mid a \in \Sigma_0\}$ and $\Sigma_2 = \{\neg\neg a \mid a \in \Sigma_0\}$ (all these new labelled events being different to each other and members of Σ_0).

Now define two partial orders on $\Sigma = \Sigma_0 \cup \Sigma_1 \cup \Sigma_2$: $\neg a <_1 a$ and $\neg\neg a <_2 \neg a$ for each $a \in \Sigma_0$, with no other pairs ordered except for those required to make τ maximal. Let us now think through the behaviour of the process

$$Probe(P) = \mathbf{Pri}_{\leq_2}(\mathbf{Pri}_{\leq_1}(P \,|||\, RUN_{\Sigma_1 \cup \Sigma_2}))$$

(i) The process inside the priority operators can perform any action of P, and also always perform any action in $\Sigma_1 \cup \Sigma_2$ without changing state.

(ii) The result of the inner priority operator can still perform any action of P and any member of Σ_2, but can now only perform $\neg a \in \Sigma_1$ when P itself is in a stable state than cannot perform a.

(iii) $Probe(P)$ can perform the same members of $\Sigma_0 \cup \Sigma_1$ as at stage 2, but can now only perform $\neg\neg a \in \Sigma_2$ when P is in a stable state which *can* perform a. When this process performs either $\neg a$ or $\neg\neg a$, its state does not change. This gives the observer, by viewing events alone, the ability to "probe" what events the current state of P can perform.

Building on this, we can for example create a stronger version of the *angelic choice* operator $\circledcirc$ of [15]: $P \circledcirc_N Q$ behaves like $P \,\Box\, Q$ except that when P and Q offer the same visible event a the choice between them is delayed rather than forced when a occurs. The operational semantics of $P \circledcirc_N Q$ can only perform a visible event a if either both P and Q perform it in parallel, or if one of them does perform it and the other one cannot.

This is implemented by running $\hat{P}\,\Theta_{\Sigma_1}\,RUN_{\Sigma_0}$ alongside $\hat{Q}\,\Theta_{\Sigma_1}\,RUN_{\Sigma_0}$. Here, $\hat{P}$ is $Probe(P)$ without the $\neg\neg a$ events, and in the combination $a \in \Sigma_0$ can synchronise with either itself or $\neg a$, in each case creating the external event a. For full details see the the extended version.

6 Comparisons

The good comparator for CSP-like operational semantics is van Glabbeek's [10] [5] concept of simply WB-cool operational semantics. This is more liberal than CSP-like because it permits cloning and because it explicitly allows arbitrary

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

probing of active arguments: it allows 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. 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. [10] 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 [4], that the use of negative premises causes no problems with the well-definedness of operational semantics.

A very detailed survey of restrictions on SOS semantics which are intended to preserve various forms of congruence is provided in [2]. This 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. The main construction in Section 5, elaborated on in the extended version of this paper, shows that the same can be done in Pri-CSP. Whereas CSP-like operators are a closed world in the sense that any composition of CSP-like operators is also describable in combinators, the same is not true of Pri-CSP-like and (n)-combinators, helping to explain why one can go beyond direct expressibility in terms of these by composing operators that are. This is why the continuation syntax T is more restricted when negative premises are allowed.

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.

In a future paper by the author and others, we will show how refinement checking over a wide variety of CSP models can be reduced, using priority, to trace refinement.

It is reasonable to ask how crucial the choice of priority is for an extra operator to achieve the degree of expressiveness seen here. 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. It cannot have a semantics in any CSP model where $\mathbf{Pri}_{\leq}(\cdot)$ does not, for it must be able to express priority. We pose this as a question for further work.

Acknowledgements

This work was done under funding from the DARPA HACMS program. It has benefited hugely from discussions with Rob van Glabbeek, Tom Gibson-Robinson, Augusto Sampaio and David Mestel.

References

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

[2] B. Bloom, W. Fokkink and R. van Glabbeek. *Precongruence formats for decorated trace semantics*, ACM Transactions on Computational Logic **5**, 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**, 863-914, 1996.

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

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

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

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

[9] 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.

[10] R.J van Glabbeek, On cool congruence formats for weak bisimulations. In *ICTAC 2005* (pp. 318-333). Springer.

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

[12] G. Lowe, *Probabilistic and prioritised models of Timed CSP*, TCS **138**, 1995.

[13] G.D. Plotkin, *A structural approach to operational semantics.* 1981.

[14] A.W. Roscoe, *The theory and practice of concurrency*, Prentice-Hall 1997.

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

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

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

[18] A.W. Roscoe, *On the expressive power of CSP extended by priority.* `http://www.cs.ox.ac.uk/files/6757/prex.pdf` Extended version of this paper.