# On the expressiveness of CSP

A.W. Roscoe

February 16, 2011

**Abstract**

We define "CSP-like" operational semantics in the context of LTS's: a severely restricted mode of describing operators which can express every operator of Hoare's CSP. Furthermore we show that every operator with CSP-like operational semantics can be simulated in CSP with the addition of an exception-throwing operator $P \,\Theta_A\, Q$ in which any occurrence of an event $a \in A$ within $P$ hands control to $Q$. Thus any language, all of whose operators are CSP-like, has a semantics over each of the behavioural models of CSP and a natural theory of refinement. This demonstrates that the extended CSP is a natural language to compile other notations into. We explore the range of possibilities for CSP-like languages, which include the $\pi$-calculus as demonstrated in a separate paper [18].

## 1 Introduction

While other languages for concurrent systems are often defined in terms of their operational semantics, the CSP approach [8, 15, 19] has always been to regard behavioural models such as *traces* $\mathcal{T}$ and *failures-divergences* $\mathcal{N}$ as 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 $\tau$.

Nevertheless CSP has a well-established operational semantics first described in SOS in [3, 5], and congruence with that is perhaps the main criterion for the acceptability of any new model that is proposed.

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 models. 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) as a natural vehicle for specification and verification.

The purpose of this paper is to devise a class of operational semantic definitions that automatically map congruently onto definitions over the 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 the next section, we remind ourselves about the CSP language and its operational semantics. In particular we recall an operator $P \Theta_A Q$ (previously described in [16]) that provides an extension in expressive power over Hoare's original CSP, which (like [19]) we include in the language. ([16] referred to the extended language as "CSP+".) As part of this exercise, we develop an intuition about what it means to be a "reasonable" operational semantics for a "CSP-like" language, and formalise it. This both develops on earlier work on CSP and is related to previous specialisations of operational semantics such as those in [1, 2, 22]. We are able to present the operational semantics of CSP in a concise notation – combinators – that is specifically designed for CSP-like languages.

The main result of this paper then follows, in which we show that any operator (or class of operators) with CSP-like, or equivalently combinator, operational semantics, can be simulated precisely in CSP. The degree of this precision depends on whether the language we are considering uses the CSP notion of successful termination $\checkmark$, and indeed we develop fully the simulation for the $\checkmark$-free case before looking at the simulation of terminating processes in the following section.

The existence of this simulation, in either case, implies that CSP-like operators have operationally congruent semantics over all CSP's models as defined in [19]. It also implies that many have the same "distributivity" property that is shared by all the standard non-recursion operators.

We can thus think of CSP as the "machine code" into which all such languages can be compiled. In Section 7 we explore the practicality of using it in this way, and look at an interesting way in which the results of this paper can guide the future development of the refinement checker FDR [14, 7].

In the following section we give some examples of operators that are CSP-like and some that are not.

A paper with this name first appeared on my web site in 2009, but this version is substantially different from that draft and in particular gives a full treatment of termination ($\checkmark$). The draft was the precursor not only of the present paper, but also [18] and the novel content of Chapter 9 of [19]. The present paper, much revised from that first one, is a companion-piece to the other two and provides their technical foundation. They, in turn, provide further motivation and examples to support the present paper.

**Acknowledgements**

I would like to thank all those who have previously asked me whether this or that construct could be modelled in CSP. For example Peter Welch has asked me many times about mobility in CSP, and Jay Yantchev inspired my work on translating Statecharts [20]. I believe that this paper shows yet again just how well chosen Tony Hoare's original CSP language was, even though we do change it a little.

I had useful discussions on this and related topics with Rob van Glabbeek during a visit to NICTA.

Thomas Gibson-Robinson, supervised by Gavin Lowe, actually implemented the simulation detailed in the first draft of this paper and in doing so provided some interesting insights into this work that have affected Section 7.

The work reported in this project was sponsored by EPSRC project ???? and grants from the US Office of Naval Research.

## 2 The operational semantics of CSP

Technically speaking, this is primarily a paper about operational semantics. Though our main result is that a wide class of languages can be modelled using the behavioural models of CSP, this will come as a corollary to our devising a translation into CSP that preserves operational semantics. We do not, therefore, need to tell the full story of what these models are and what they are all good for. Formally speaking, what we mean by "preserves operational semantics" here is that the resulting process has semantics that are strongly bisimilar to the original, or to an algebraically transformed original when dealing with the termination event $\checkmark$. Whenever we refer to bisimulation in this paper we mean strong bisimulation.

Readers interested the abstract, behavioural models of CSP should consult [15, 17, 16, 19], and in particular the last of these. Perhaps the most important are the simple *finite traces* model $\mathcal{T}$ and the *failures divergences model* $\mathcal{N}$, the latter with the addition of infinite traces $\mathcal{U}$ in the case that unboundedly nondeterministic operators are used. These respectively allow us to specify safety properties and finitary liveness ones (e.g. *after trace $s$, the process will definitely accept an event from the set $X$ if it is offered*).

The language we use in this paper was termed CSP+ in [16], because there is an extra operator that strictly extends its expressive power beyond the language described by Hoare [8]. However, in the spirit of incrementation and for consistency with [19], we now use the original name for the extended language.

We now present a new notation for expressing the operational semantics of CSP (also used in [19]), while developing an intuition for what it means to be "CSP-like".

The operational semantics of CSP creates an LTS, and indeed it is natural to interpret free process variables as taking values in some ground LTS. This type of operational semantics is traditionally presented within the Structured Operational Semantics, or SOS style. A complete SOS operational semantics for CSP can be found in [15]. That book, whose operational semantics we regard as standard, will be abbreviated by its initials TPC in the rest of this paper.

The actions of the LTS fall into three categories: ordinary visible actions $a \in \Sigma$ which can only happen when the observer agrees to them, the invisible internal action $\tau$ and the termination signal $\checkmark$. Following [15, 19], we treat this as an observable but uncontrollable *signal* that a process performs to indicate that a process has terminated. When $\checkmark$ is observed it always represents the end of an observation: for convenience we introduce a special unobserved state $\Omega$ to which all $\checkmark$ actions lead.[1]

In SOS style we need rules to infer every action that each operator can perform when applied to its appropriate number of arguments. Typically, but not invariably, these actions result from one or more of these arguments performing actions, as with the external choice operator $\Box$ which gives the environment the choice between the initial visible actions of the two arguments.

In some sense the main rules for this operator say that the first $\Sigma$ action that either process performs resolves the choice:

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

[1]While we clearly need to handle the event $\checkmark$ if we are to give a complete theory for CSP itself, it is likely that, like $\pi$-calculus [10, 18], some languages that our theory applies to (i.e. CSP-like ones) will not require $\checkmark$. As will be apparent from the rest of this paper, that leads to a significant simplification of the theory and a strengthening of the results.

This means that $\Box$ needs to have both its process arguments "turned on" to allow them to perform these actions. There is an important distinction between arguments that are **on** like these and ones that are **off**: not yet activated so that no immediate actions can depend on them. In the first case it would be reasonable to stipulate that any action (specifically $\tau$ or $\checkmark$) that an observer cannot stop an active process from performing cannot be prevented: if $P$ can perform one of these actions on its own account then so can it in any context like $P \Box Q$ where it is **on**. We must therefore expect that there will be SOS rules to say what $P \Box Q$ does when $P$ or $Q$ performs $\tau$ or $\checkmark$. Furthermore we must also expect that the actions that result from these rules will be ones that an external observer cannot refuse – otherwise the operator, by extension, would have to be able to prevent the **on** argument from performing $\tau$ or $\checkmark$. So the result action must, in these cases, be $\tau$ or $\checkmark$. In fact the SOS rules for $\Box$ are

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

$$\frac{P \xrightarrow{\checkmark} P'}{P \Box Q \xrightarrow{\checkmark} \Omega} \qquad \frac{Q \xrightarrow{\checkmark} Q'}{P \Box Q \xrightarrow{\checkmark} \Omega}$$

There is an important difference between the first pair of rules and the corresponding rules for the $+$ operator of CCS: a $\tau$ here does not resolve the choice. If $P$ or $Q$ performs a $\tau$ then the overall pattern of $P \Box Q$ remains completely unchanged. It is easy to argue that this is natural: if an observer of $P$ cannot see internal actions $\tau$, then neither can an operator being applied to $P$. In fact every CSP operator follows this principle: whenever an argument is **on**, that argument is allowed to perform a $\tau$ without changing the overall state of the operator. The $\tau$ is simply *promoted* to the outside as in the rules above.

That this is true is a reflection of the fact that in the theory of CSP, in all the behavioural models used for denotational semantics, a process $P$ is always equivalent to the process $\tau.P$ (as it would be written in CCS) that performs an initial $\tau$ action.

It follows that, as long as we define which arguments of a CSP (or *CSP-like*) operator are **on**, there is no need to give the $\tau$ promotion rules explicitly. We do need to give $\checkmark$ rules, however, since the result of an argument performing a $\checkmark$ can be either $\checkmark$ or $\tau$. In the first of these cases there is (as with $\tau$-promotion) no choice about the form of the successor state, but in the second case there is. Consider the sequential composition operator

$P;\ Q$. Only its left-hand argument is **on**, and so has an implicit $\tau$ rule. The other rules allow the left-hand argument to perform any $\Sigma$ action, and to terminate (promoted to $\tau$) and start up the second argument:

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

In general it would have been reasonable to specify that, when one argument terminates, that termination becomes a $\tau$ at the high level and the process continues as any allowable CSP combination of the remaining arguments. We will come back to discuss the word "allowable" shortly.

When **on** arguments are allowed to perform actions in $\Sigma$, the resulting action might be the same, as in the $\Box$ case above, a different visible action as in renaming

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

or $\tau$ as in hiding

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

[There is a further rule for $P \setminus X$ covering the case of an action $a \notin X$: this is simply promoted as itself.] There is no reason why an action $a \in \Sigma$ in an argument should not give rise to an externally visible $\checkmark$, but in fact no standard CSP operator has this effect.

In all the examples we have seen so far there is a one-to-one relationship between the actions of the argument processes and those of the whole construct. In one sense this is true in general: each argument action that is permitted will give rise to one result action. However in general there may not be a single argument performing an action that underlies an external one: there may be none at all as in nondeterministic choice

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

and prefixing:

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

In each of these cases an operator performs actions when all its arguments are **off** and then turns on one of these.

Alternatively an operator can force several **on** arguments each to generate an action, which it synchronises into a single externally visible one, as in the parallel operator:

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

(where there are also clauses that allow either $P$ or $Q$ to perform $a \in \Sigma \setminus X$ independently of the other).

A formulation that works for all actions of CSP operators other than those produced by arguments' $\tau$s and $\checkmark$s, is as follows:

- Any action $x$ of an operator $F(P_1, \ldots, P_n, \mathbf{Q})$, whose **on** arguments are the $P_i$, may be enabled by all members of a (perhaps empty) subset of the $P_i$ each performing some chosen $a = \phi(i)$. ($\phi$ is thus a partial function from the **on** indices to $\Sigma$, with $dom(\phi)$ being the set of arguments participating in this synchronisation.)

- In any state where all the members of this subset $dom(\phi)$ can perform the respective $\phi(i)$, the component arguments synchronise and generate $x$.

- The result state is any allowable process expression in terms of the arguments $P_i$ and $Q_\lambda$, in which whenever a $P_i$ appears with $i \in dom(\phi)$, the process involved is $P_i'$, the result of $P_i$ performing $\phi(i)$.

In the cases of standard CSP operators, all the result states are *either* a single argument (as in $\Box$ above) or retain exactly the same format as before the action (as in hiding, renaming, parallel and the non-$\checkmark$ case of $P; Q$). We can, however, broaden this somewhat but not arbitrarily. So let us consider what a process expression that follows an action (i.e. determining the structure of the successor state) should look like:

- It can contain any subset of the original arguments, discarding the others.

- There is no restriction on how an **off** argument can be used: it may still be **off**, be turned **on**, or indeed appear in both sorts of places.

- However if an **on** argument is present, it must still be **on** and *cannot be copied*: in other words it must appear only once.

For example, operators whose operational semantics contained the clauses

$$\frac{P \xrightarrow{a} P'}{\oplus(P) \xrightarrow{a} P' \underset{\Sigma}{\parallel} P'}$$

$$\frac{P \xrightarrow{a} P'}{P \otimes Q \xrightarrow{a} Q;\ P'}$$

would not be CSP-like as these result states would not be allowable in the above sense.

There is a fundamental reason why the first of these cannot be allowed: such an operator would not be consistent with the usual behavioural models of CSP in which each process is represented by a set of linear observations. This representation, as discussed for example in [19], is intimately connected with the principle that operators must be *distributive* over nondeterministic choice $\sqcap$ in their **on** arguments, namely

$$\begin{array}{l} \oplus(P_1 \sqcap P_2) \\ = \oplus P_1 \sqcap \oplus P_2 \end{array}$$

in this case. This would demand that

$$\oplus(a \to a \to STOP \sqcap a \to b \to STOP) = \oplus(a \to a \to STOP) \sqcap \oplus(a \to b \to STOP)$$

The operational semantics above would not make this true, since the left-hand side can deadlock after the trace $\langle a \rangle$, whereas the right-hand side cannot. It has been known for many years that CSP models cannot handle operators which can “clone” arguments in flight, and this restriction just prevents us from defining such operators via operational semantics.

No CSP construct can suspend the execution of an argument – moving it from a state where it is active to one where it is as though it has still to be turned on and in particular is no longer capable of performing $\tau$s. The theoretical problems that allowing such behaviour to be termed CSP-like would cause are less than with copying[2]. However complex conditions would be needed to ensure that no copying could be allowed in by the back

[2]Indeed, a semantics in all CSP models could be given to an operator $suspend_a(P)$ which behave like $P$ until the event $a$ (normally outside $P$'s alphabet) occurs, at which point the only possible action is a further $a$ which re-starts $P$ in the same state where it left off, before a further $a$ and so on. During suspension the suspended $P$ is prevented even from performing $\tau$ and $\checkmark$ actions. To the author there seems a much less persuasive argument for including this and similar than there is for $\Theta_A$.

door if we did allow suspension. And there seems to be no overriding reason for allowing such behaviour.

Notice here that we make no restrictions about how many times an **off** argument of $OP$ can be used. This is perhaps as well, since we should observe that constructs such as

$$\mathit{Farm}(P) = \mathit{start} \rightarrow (P \underset{\emptyset}{\|} \mathit{Farm}(P))$$

can actually copy the argument $P$ arbitrarily often. (This one just interleaves a "farm" of as many $P$s as *start* events have occurred – assuming that *start* is not an event of $P$.) There is no theoretical objection to applying this type of construction to a process that has yet to be turned on. One could give a direct operational semantics to this construct via the family of operators $\mathit{Farm}_n(Q_1, \ldots, Q_n, P)$ which runs $Q_1$ to $Q_n$ in parallel (all as **on**) and also offers *start* independently of its arguments before moving to a $\mathit{Farm}_{n+1}$ state. Thus $P$ is an **off** argument: $\mathit{Farm} = \mathit{Farm}_0$, where

$$\frac{}{\mathit{Farm}_n(Q_1, \ldots, Q_n, P) \xrightarrow{\mathit{start}} \mathit{Farm}_{n+1}(Q_1, \ldots, Q_n, P, P)}$$

$$\frac{Q_i \xrightarrow{x} Q_i'}{\mathit{Farm}_n(Q_1, \ldots, Q_i, \ldots, Q_n, P) \xrightarrow{x} \mathit{Farm}_n(Q_1, \ldots, Q_i', \ldots, Q_n, P)}$$

## 2.1 Combinator operational semantics

The above essentially defines what it means for an operational semantics to be CSP-like, but we can give a clearer picture with a new notation, in terms of *combinators*. To specify the operational semantics of an operator $F(\mathbf{P})$ you first define which of the arguments is **on**. For convenience we will assume that these are $P_1, \ldots, P_m$ for some $m \geq 0$ and that the **off** ones are $\mathbf{Q} = \langle P_i \mid i \in I \rangle$ for some $I$ disjoint from $\{1 \ldots, m\}$.[3] There is no need to write down anything further for the $\tau$ promotion rules of the operator we are defining: we know that if $P_i$ is an **on** argument then $P_i \xrightarrow{\tau} P_i'$ then

$$F(P_1, \ldots, P_i, \ldots, P_m, \mathbf{Q}) \xrightarrow{\tau} F(P_1, \ldots, P_i', \ldots, P_m, \mathbf{Q})$$

There is an arbitrary collection of $\Sigma$-rules, ones where the $P_i$ perform some pattern of actions in $\Sigma$. These take one of two forms. A rule of the form

[3]As we will show later, there can be no reasonable operator with infinitely many **on** arguments. However there is no reason why there cannot be infinitely many **off** ones, and CSP itself has two examples of this, namely some cases of nondeterministic choice and prefix choice.

$((x_1, \ldots, x_m), y)$, where each $x_i$ is either[4] in $\Sigma$ or is $\cdot$, and $y \in \Sigma \cup \{\tau, \checkmark\}$ says that when each of the $P_i$ that has $x_i \in \Sigma$ performs $x_i$ and enters state $P'_i$, the combination performs $y$ and (unless $y$ is $\checkmark$) enters state $F(P''_1, \ldots, P''_n, \mathbf{Q})$, where $P''_i = P_i$ if $x_i = \cdot$, and $P'_i$ if $x_i \in \Sigma$. Thus a rule of this form covers every case (like the SOS rules of $\|$, $\setminus X$ and $[\![R]\!]$ quoted above) where a collection of the arguments synchronise on a pattern of actions and produce $y$, but where the overall shape of the program does not change. In the case where $y = \checkmark$ we conventionally assume that the result process is $\Omega$, as with similar SOS rules.

The second form covers the case where the overall shape of the program does change. These have an extra component, namely a piece of syntax $T$ in which arguments are represented as bold-face indices drawn from $\{1, \ldots, m\} \cup I$, so $\mathbf{1}$ represents the first argument, and so on. Thus $((x_1, \ldots, x_m), y, T)$, where $x_i$ and $y$ are as before, has the same effect as $((x_1, \ldots, x_m), y)$ except that 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 **on** index $\mathbf{i} \in \{1 \ldots, m\}$ can appear more than once in $T$.
- Such **on** 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 **on**, and are never nested within inner recursions). 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 $\bigoplus(\ldots, T, \ldots, \mathbf{Q})$, where the place $T$ occurs at an **on** place in the arguments of the CSP-like operator $\bigoplus$.
  - No other appearance of $\mathbf{i}$, including any in a recursive definition, is ID.

[4]Note that these finite tuples of "preconditions" for the rule are simply a convenient notation for the partial functions $\phi$ refered to above.

> It is worth remarking that these pieces of syntax can contain arbitrary *closed* CSP processes at any point without restriction[5]. 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 $\Sigma$-rules $(a,a)[a \notin X]$ and $(a,\tau)[a \in X]$, using the convention that for operators like this one with a single **on** 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 $P \,\square\, Q$ does change the process structure, so its $\Sigma$ rules are

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

indicating that either side can perform any action in $\Sigma$ which resolves the choice, eliminating the $\square$. These rules have exactly the same meaning as the SOS ones for $\Sigma$ actions in $\square$ given above.[6]

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 $P \,\square\, Q$:

$$((\checkmark,.),\checkmark) \qquad \text{and} \qquad ((.,\checkmark),\checkmark)$$

Rules in which the output action is not $\checkmark$ always have it $\tau$, as in the $\checkmark$-rule for $P;\ Q$ where only the first argument is **on** and we give $Q$ the index $-\mathbf{1}$:

$$(\checkmark,\tau,-\mathbf{1})$$

Again this means exactly the same as the corresponding SOS rule above. Since these cases of a $\checkmark$ leading to a $\tau$ always discard the terminating argument, such rules always have to give the syntax of the successor process explicitly.

[5]In fact it is easy to show that any LTS at all is the operational semantics of such a process, using a straightforward infinite mutual recursion and hiding to create any $\tau$s.

[6]As in SOS, when we present a combinator operational semantics the rules we present are implicitly quantified over members of $\Sigma$, sometimes subject to side conditions. Thus a single syntactic rule scheme such as the above may represent many actual transition rules.

## 2.2 Combinator semantics for CSP

We have already given the full combinator semantics for $\Box$ above. The only rule that remains for hiding is the $\checkmark$ one: $(\checkmark, \checkmark)$ because the termination of the only argument causes the whole to terminate. Renaming[7] $P[\![R]\!]$, where agains the only argument is **on**, has exactly the same termination rule and the single $\Sigma$ rule $(a, b)[a\,R\,b]$.

We have seen the termination rule for $P;\ Q$, where only the first argument is **on**. There is one $\Sigma$ rule, namely $(a, a)$ since $P$ can perform any non-$\tau$ action without changing the overall state, as set out in SOS above.

Another operator that has one argument **on** and one **off** is the (asymmetric) sliding choice operator $P \rhd Q$: only the first argument is **on**. There are three rules:

$$(a, a, \mathbf{1})\ [a \in \Sigma] \qquad (\cdot, \tau, -\mathbf{1}) \qquad (\checkmark, \checkmark)$$

where again the second and **off** argument has index $-\mathbf{1}$. The most interesting rule here is the second, since it enables an action (here $\tau$) that is unrelated to any action of the **on** process: $P \rhd Q$ can offer actions of $P$ and be resolved by a visible action of $P$, but until it is resolved in this way can always be resolved in favour of $Q$ by performing a $\tau$. This last $\tau$ switches $Q$ **on**.

Both arguments of the parallel operator $\underset{X}{\|}$ are **on** and it has the following three rules for its $\Sigma$ actions:

$$((a, \cdot), a)\ [a \notin X] \qquad ((\cdot, a), a)\ [a \notin X] \qquad ((a, a), a)\ [a \in X]$$

or, in other words, the two arguments progress independently when not communicating in $X$, but have to synchronise over $X$.

Recall that $P \underset{X}{\|} Q$ terminates when both $P$ and $Q$ do. Following the rules for termination above, however, it would not be proper to insist on the two $\checkmark$s synchronising. Therefore, isomorphic to the SOS treatment of

[7]There is a convention, sometimes stated explicitly and sometimes not, that the renaming relations used in CSP are total on the events used by the process to which it is being applied. Thus if $P \xrightarrow{a} P'$ then there is at least one $b$ with $a\,R\,b$. Renaming makes sense without this restriction: events $P$ attempts to perform not in $domain(P)$ are simply blocked, and that is the meaning of both the SOS and combinator versions of transition rules for this case. We will allow this generalisation in this paper, because it makes some of the (already complex) process descriptions later more concise. Any such extended use of renaming is equivalent to a version that does not need the extension, for example $(P \underset{\Sigma \setminus dom(R)}{\|} STOP)[\![R]\!]$ in the case where $P$ never terminates.

this operator in TPC, we make both the processes' ✓s into $\tau$s and follow these with an externally visible one. In the following we regard the syntaxes $P \underset{X}{\|} \Omega$ and $\Omega \underset{X}{\|} P$ as separate unary operators on $P$. The termination rules are then

$$((\checkmark, \cdot), \tau, \Omega \underset{X}{\|} \mathbf{2}) \quad \text{and} \quad ((\cdot, \checkmark), \tau, \mathbf{1} \underset{X}{\|} \Omega)$$

where the rules for both of the unary constructs $P \underset{X}{\|} \Omega$ and $\Omega \underset{X}{\|} P$ are

$$(a, a)\,[a \notin X] \quad \text{and} \quad (\checkmark, \tau, \mathit{SKIP})$$

We could also have given a semantics where the second ✓ is passed to the outside, rather than becoming a $\tau$ that leads to *SKIP*. That would not be isomorphic to TPC, though.

In interrupt, $P \triangle Q$, both arguments are **on**. The left-hand argument can perform any series of actions but at any time, $Q$ can perform any visible action and take over unless $P$ has already terminated. Its the rules are

$$\begin{array}{cc} ((a, \cdot), a)\,[a \in \Sigma] & ((\cdot, a), a, \mathbf{2})\,[a \in \Sigma] \\ ((\checkmark, \cdot), \checkmark) & ((\cdot, \checkmark), \checkmark) \end{array}$$

It seems a little strange that the second argument of $\triangle$ should be **on**, but of course this is necessary for it to be able to perform an initial event which instigates the interruption. Of course in $P \triangle (a \to Q)$, a very common pattern of use, $Q$ is initially **off**.

This completes the semantics of the established CSP operators apart from one important class, namely those with no **on** arguments at all. These include the various constant processes, prefixing and nondeterministic choice. For each of these we represent the empty tuple of actions from the non-existent **on** arguments as $^-$ and use negative integers as the indices of **off** arguments.

- *STOP*: no rules
- *SKIP*: $({}^-, \checkmark)$
- $P \sqcap Q$: $({}^-, \tau, -\mathbf{1})$ and $({}^-, \tau, -\mathbf{2})$
- $a \to P$: $({}^-, a, -\mathbf{1})$

Notice that we have only covered the binary form of $\sqcap$ and the simple form of prefix here. That is because general forms of each of them can have any number – even an infinite number – of **off** arguments.

In such cases we have to be careful to define the indexing for the families of **off** arguments that these operators have. Thus the operator $\bigsqcap\{P_i \mid i \in I\}$, whose arguments are parameterised by $I$, will simply have the rules

$$\{(^-, \tau, \mathbf{i}) \mid i \in I\}$$

and the general prefix $e \rightarrow P$ has arguments parameterised by $comms(e)$ and rules[8]

$$(^-, a, \mathbf{a})\ [a \in comms(e)]$$

in which the argument labelled $\mathbf{a}$ is $subs(e, a, P)$.

Notice that even when they are infinite in number, only one of the arguments ever becomes turned **on** in a run of prefix or nondeterministic choice. It is not permissible to extend the notation so that there are an infinite number of **on** arguments for any operator. It seems plausible that we could introduce an infinite form of $\Box$, again parameterised by $I$, whose rules (choosing to represent the partial functions as sets of pairs rather than infinite tuples) would be

$$\{(\{(i, a)\}, a, \mathbf{i})\ [a \in \Sigma] \mid \lambda \in I\} \cup \{(\{(i, \checkmark)\}, \checkmark) \mid i \in I\}$$

At first sight this seems uncontroversial and straightforward, but promoted $\tau$ actions create a problem: imagine combining together infinitely many

[8]As is also true for the corresponding SOS treatment of prefixing, we would be able to handle general prefix and generalised nondeterministic choice much more elegantly if we were to add semantic environments that bind free identifiers to their values in the combinator operational semantics. In fact doing so would give an even clearer distinction between **on** and **off** arguments, as an **on** one must already have its environment and an **off** one need not. There would still be the choice of whether to show the environment explicitly in the semantic term. Of course if we did so then the appearance of all the operational semantic clauses would change. However, at least for the purpose of giving a semantics to CSP, we can use an implicit notation where it is assumed that the environment given to each newly turned-**on** argument is the same as the "input" one unless we state a modification explicitly. In this, we might write the rule for prefixing as

$$(^-, a, (\mathbf{1}, subs(e, a, \rho)))\ [a \in comms(e)]$$

In other words, we now treat prefixing as a unary operator and use $subs(e, a, \cdot)$ in a modified form on a conventional name $\rho$ for the surrounding environment. In this style we could have two different infinitary forms of $\bigsqcap$: one with an infinite set of processes, and the other with an infinite set of (perhaps tuples of) values to be substituted into the environment.

$STOP$ processes, and infinitely many $STOP \sqcap STOP$ processes. Since $STOP = STOP \sqcap STOP$ in every CSP model, the results ought to be the same. But they are not, since the $\tau$ actions at the start of the second process could all be promoted, one after another, through an infinite $\Box$, creating a divergence and eliminating stability. Thus if we were to allow an infinite form of $\Box$ or any other operator with infinitely many **on** arguments it could not be congruent over any model for CSP other than the traces model $\mathcal{T}$. We can be confident of this thanks to the fact that, by the results of [19] (Chapters 11 and 12) every CSP model other than $\mathcal{T}$ refines either the stable failures model $\mathcal{F}$ or the divergence-strict trace model $\mathcal{T}^{\Downarrow}$. In neither of these models is the operational semantics of the external choice $\Box$ of an infinite number of $STOP$ processes congruent to the same choice of an infinite number of $STOP$s preceded by a single $\tau$, even though the latter process is equivalent to $STOP$ in all CSP models.

There is a further operator, not a traditional part of CSP but which plays both an important theoretical role in this and other works [16, 19] and which has been seen in [19] to have a number of interesting practical uses. This is the *throw* operator which models a process throwing an exception, with a behaviour to follow subsequent to this: $P \,\Theta_A\, Q$ acts like $P$ until the latter performs an event in $P$, whereupon it discards $P$ and acts like $Q$. Thus $P$ is an **on** argument and $Q$ (index $-\mathbf{1}$) is initially **off**. It has the $\Sigma$ rules

$$(a, a)\,[a \notin A] \quad \text{and} \quad (a, a, -\mathbf{1})\,[a \in A]$$

and the simple $\checkmark$ rule $(\checkmark, \checkmark)$.

Hopefully it is clear that combinator rules for the operational semantics of CSP are extremely concise and intuitive, while at the same time capturing implicitly at least much of what we meant by "CSP-like" earlier.

That in itself would probably not be sufficient reason to present the operational semantics in a non-standard way, as we have. The real motivation is that we will, in the next sections, be able to show that *any* operator that has a combinator operational semantics can be translated naturally into CSP and therefore makes sense over every denotational model of CSP.

In other words, any operator defined in this way that is not already a CSP operator is a natural extension to CSP.

**Definition 1** *A CSP-like operator is one over LTSs which can be defined by combinator operational semantics.*

We have not yet given the operational semantics of recursion. The term $\mu\, p.P$ (the recursive solution to the equation $p = P$, where $p$ is a process variable that can appear in the CSP term $P$) has two alternative operational semantics as discussed in [15, 19]. The first is to say, in SOS, that

$$\frac{}{\mu\, p.P \xrightarrow{\tau} P[\mu\, p.P/p]}$$

or in other words that unwinding recursion generates a $\tau$ action which expands the term. This is completely unproblematic from the point of view of evaluating the transitions of processes: when combined with the rest of the operational semantics it yields a unique solution for the initial actions of every term. It does, however, introduce $\tau$ actions that are frequently inconvenient both practically and, in the present paper, theoretically.

The alternative is to re-write $\mu\, p.P$ to $P[\mu\, p.P/p]$ *without* introducing the additional $\tau$ action. In fact the extra understanding brought by our analysis of **on** and **off** arguments lets us understand exactly when this is unproblematic: it is when every occurrence of $p$ within the term $P$ occurs in an **off** position in $P$, where an **off** position in $P$ means intuitively that the term $P$ does not depend on the initial actions of $p$ to determine its own initial actions.

This can easily be calculated recursively:

- A process variable term $p$ depends directly on itself but on no other process variable.
- The term $F(P_1, \ldots, P_n, \mathbf{Q})$, where $P_i$ are the **on** arguments of $F$, depends directly on all process variables that the $P_i$ do.
- The recursion $\mu\, p.P$ is allowable if $P$ does not depend directly on $p$. In that case $\mu\, p.P$ does not itself depend on $p$. It depends directly on other process variables if and only if $P$ does.
- The mutual recursion $\underline{p} = F(\underline{p})$ is allowable if and only if the relation on the indexing set $\Lambda$ in which $\lambda > \mu$ if $P_\lambda$ depends on $p_\mu$ is a well-founded partial order.

  In this case the process denoted by $p_\lambda$ depends on just those process variables $q$ that are outside $\underline{p}$ and upon which one of the $P_\mu$ with $\lambda \geq \mu$ depends.

As has frequently been remarked, essentially all *sensible* CSP definitions have all their recursions allowable in this sense, meaning that, as FDR does, it makes practical sense to adopt the $\tau$-free operational semantics. In doing so it is really necessary to restrict ourselves only to use the allowable recursions defined above. In FDR this is necessary because asking it to compile a non-allowable recursion will mean that FDR itself does not terminate. While in theory we could still define the operational semantics of a non-allowable term such as $\mu\, p.p$ to be those transitions that are positively derivable from the inference rules implied by combinators and unwinding, this would not correspond to the intended CSP semantics. $\mu\, p.p$ would have no transitions, whereas CSP semantics demands that it diverges (i.e. performs an infinity of $\tau$s), as in the $\tau$-unwinding semantics of this recursion.

It follows that if we are to adopt the $\tau$-free recursion syntax, they should be restricted to satisfy the above conditions.

## 2.3 Comparisons and background

CSP-like operational semantics, as presented here, have many precursors. Many of the ideas presented here, such as the difference between **on** and **off** arguments, the promotion of $\tau$s, and the requirements for successor states, were originally discovered by the author in his own doctoral thesis [13] (Chapter 8) about 30 years ago, before SOS started to dominate his view of what an operational semantics looks like. It was, however, the results of the rest of this paper that persuaded him to re-work the formal presentation of operational semantics in this way: a form of operational semantics that corresponds exactly with CSP.

In the interval since then, a number of authors, notably Bloom and van Glabbeek, have investigated conditions on SOS semantics that induce congruences (for the operators they define) with respect to a large number of forms of equivalence on transition systems, such as branching and weak bisimulation and a number of individual CSP-based models. This has led them to rediscover the ideas outlined above (with $\tau$ promotion being referred to as *patience* rules, and **on** and **off** being *fluid* and *frozen.* Many of the conditions we have expressed here, for example on the form of successor terms, are directly analogous to ones used in that work. For example in [2] it is shown that restrictions on operational semantics which are essentially the same as ours but without the no-copy rule ensure that operators are a congruence with respect to weak bisimulation. These papers are typically highly technical and explore the consequences of many alternative decisions that might have been made.

We have here aimed to concentrate on a single clear structure of operational semantics that discovers CSP's core structure, and which can lead to the results set out in the following sections, and to creating a notation which supports this structure.

There are two particular ways in which the formulation here goes beyond earlier ones: firstly the careful treatment of the termination signal $\checkmark$, and the fact that we allow an infinite family of **off** arguments, following the CSP language itself.

FDR has, to all intents and purposes, used combinators to implement the state machines explored in its main running phase since the advent of FDR 2 in 1995. We will examine the way it does this in a little more detail at the end of the next section.

## 3 A re-working of combinators

The form in which combinators were represented above gives (at least in the author's opinion!) a very natural way to present an operational semantics, where it is sufficiently rich. On the other hand the way we have allowed successor states to be represented as pieces of syntax does not fit well with the forms of simulation we will be developing later. Therefore in this section we will give a somewhat lower-level representation of combinators that can be derived straightforwardly from the notation above.

Given any immediately distributive term $t$, formed from a finite family of **on** arguments and an indexed family of **off** ones, it is possible to derive direct combinator rules for that combination from the ones of its constituent operators (together with the implicit $\tau$-promotion rules these operators have). These can be derived inductively over the syntax of such a term, with the fact that each **on** argument appears at most once in $t$ being crucial. Suppose $t = \bigoplus(t_1, \ldots, t_m, \mathbf{S})$, where $\mathbf{S}$ is a family of terms that do not depend on the **on** arguments $\mathbf{1}, \ldots, \mathbf{n}$, each of which appears in exactly one of the $t_i$.

Then inductively we may assume that each of the $t_i$ has a combinator semantics.

It is clear that any $\tau$ action of an **on** argument is promoted to a $\tau$ of one of the $t_i$, and in turn by $\bigoplus$ since it is CSP-like. Similarly any $\checkmark$ of an **on** argument is promoted to a $\tau$ or $\checkmark$ of one of the $t_i$, and hence to a $\tau$ or $\checkmark$ of the whole term by the corresponding $\checkmark$ or $\tau$ promotion rule of $\bigoplus$. In the case where $\checkmark$ maps to $\tau$, the form of the successor process is determined at the syntactic level where the transformation from $\checkmark$ to $\tau$ occurs: if it is within a $t_i$ (which transforms to $t_i'$) then the final form is $\bigoplus(t_1, \ldots, t_i', \ldots t_n, \mathbf{S})$. Otherwise, when a $\checkmark$ of $t_i$ is transformed to $\tau$ by $\bigoplus$, the final form is derived from a $\checkmark$ rule of $\bigoplus$ for argument $\mathbf{i}$.

Similarly any $\Sigma$-rule of any $t_i$ that yields a $\tau$ action is promoted to a $\Sigma$ rule of the whole term that yields a $\tau$: the $i$th argument of $\bigoplus$ being transformed exactly as in $t_i$, the result of the structure of $t$ being unchanged.

The $\Sigma$-rules of $\bigoplus$ that have preconditions $\phi$ (with domains subsets of $\{1, \ldots, m\}$, corresponding to $\bigoplus$'s arguments), can be combined with any combination of the rules $r_i$ of the $t_i$ for $i \in dom(\phi)$ such that $x_i = \phi(i)$ (i.e. the output action $x_i$ of $r_i$ is $\phi(i)$). This gives a new rule whose precondition is the union[9] of those for the $r_i$, and whose output action is that of the top-level rule. The successor state is formed by the top-level rule from

- **S** (its **off** arguments)
- the $t_j$ such that $j \notin dom(\phi)$
- $t_i'$, the successor states of the rules used for $t_i$ $(i \in \mathrm{dom}(\phi))$.

Notice that the successor term calculated above is itself always immediately distributive.

For example, consider the term $\mathbf{1} \bigtriangleup ((\mathbf{2};\ Q) \;\Box\; \mathbf{3})$ with three **on** arguments and one **off** one $(Q)$. We can build combinators for this from those of the constituent operators.

- $\checkmark$ from **1** or **3** can terminate the whole, the latter from rules of $\bigtriangleup$ and $\Box$ combined. This gives the rules $((\checkmark, \cdot, \cdot), \checkmark)$ and $((\cdot, \cdot, \checkmark), \checkmark)$.
- $\checkmark$ from **2** starts the second argument of **2**; $Q$ and generates a $\tau$ that is promoted from the first **on** argument of $\Box$ and the second of $\bigtriangleup$, giving the rule $((\cdot, \checkmark, \cdot), \tau, \mathbf{1} \bigtriangleup (Q \;\Box\; \mathbf{3}))$. This turns $Q$ **on**.
- $a \in \Sigma$ can happen in **1** without changing the overall pattern or being changed, giving the rule $((a, \cdot, \cdot), a)$. This is based on the rule $((a, \cdot), a)$ of $\bigtriangleup$.
- $a \in \Sigma$ can happen in **2** without changing the pattern of **2**; $Q$ while resolving both of the higher level operators. This combines the rules $(a, a)$ of ;, $((a, \cdot), a, \mathbf{1})$ of $\Box$ and $((\cdot, a), a, \mathbf{2})$ of $\bigtriangleup$ to give the rule $((\cdot, a, \cdot), a, (\mathbf{2};\ Q))$.
- Similarly we get the rule $((\cdot, \cdot, a), a, \mathbf{3})$ to show $\Sigma$ actions of the third argument resolving both choices.

Thus the successor term in every combinator rule can be thought of as a single CSP-like operator applied to a selection of arguments and closed processes rather than a possibly complex term. Note that in cases like

[9]Note that this union construction is unproblematic because of the restriction that each **on** argument **i** appears in exactly one of the $t_j$.

the last rule above where the result is just one argument, we can think of it as the identity operator *id* (whose rules are $(a, a)$ and $(\checkmark, \checkmark)$) applied to that argument. This term satisfies the condition for being immediately distributive itself.

We can simplify this further, since in every instance where an operator introduces a closed term as an operator argument, we can add that term as an additional (implicit, perhaps) **off** argument of all operators from which this one can appear as a state.

Given a rule of operator $OP_\alpha$ with "arity" $(n(\alpha), I(\alpha))$ (of **on** and **off** arguments respectively) that results in a successor term whose operator is $OP_\beta$, we can completely describe the final state once we know how the arguments of $OP_\alpha$ are re-arranged to form those of $OP_\beta$.

- The **off** arguments of $OP_\beta$ are all **off** arguments of $OP_\alpha$. They are therefore determined by a function from $I(\beta)$ to $I(\alpha)$.
- The **on** arguments of $OP_\beta$ are either **off** or **on** arguments of $OP_\alpha$. They are therefore determined by a function from $\{1, \ldots, n(\beta)\}$ to $\{1, \ldots, n(\alpha)\} \cup I(\alpha)$ (recalling that the last union is disjoint) with the property that no $j \in \{1 \ldots, n(\alpha)\}$ is the image of more than one $i$. This last restriction is to prevent the successor process having more than one instance of an **on** argument.

If we analyse what happens to the arguments of $OP_\alpha$, some of them can be discarded, **off** and **on** arguments preserved, and **off** arguments turned **on**. It is possible that a given **off** argument may be preserved as an **off** one and/or turned **on** several times. In other words there is no reason why **off** arguments cannot be copied.

It follows that an arbitrary $\Sigma$ rule $\rho$ of $OP_\alpha$ is completely described by a tuple $(\phi, x, k, f, \psi, \chi)$ where

- $\phi$ is a partial function from $\{1, \ldots, n(\alpha)\}$ to $\Sigma$. Its meaning is that, in order for this transition to fire, each argument $P_j$ such that $j \in dom(\phi)$ must be able to perform the action $\phi(j)$ and become some $P'_j$. Note that this imposes no condition if $dom(\phi)$ is empty.
- $x$ is the action in $\Sigma \cup \{\tau, \checkmark\}$ that $OP_\alpha(\mathbf{P}, \mathbf{Q})$ performs as a consequence of the condition expressed in $\phi$ being satisfied.
- $\beta$ is the index of the operator (i.e. in the extended class of CSP-like operators that includes immediately distributive combinations of

basic ones) that forms the result state of this action. For clarity, in the examples below we will usually replace this index with our name for the operator itself, or syntax of the same sort we used when giving a notation for combinators.

- $f$ is a total function from $\{1, \ldots, k\}$ for some $k = k(\rho) \geq 0$ to $I(\alpha)$ that represents, in some chosen order, the indexes of the components of **Q** (i.e. the **off** arguments), that are started up when the rule fires.
- $\psi : \{1, \ldots, n(\beta)\} \to \{1, \ldots, n(\alpha) + k(\rho)\}$ is the (total) function that selects each of the resulting state's **on** arguments. It must be injective and include the whole of $\{n(\alpha)+1, \ldots, n(\alpha)+k(\rho)\}$ in its range. This says that no **on** argument of $OP_\alpha$ can be cloned and that the newly turned on processes are used once each.
- $\chi : I(\beta) \to I(\alpha)$ is the total function that selects the **off** arguments of $OP_\beta$. Since there is no requirement that $f$ is injective, the **off** arguments can be copied at this stage.

The $\checkmark$ rules can be written in a similar style, the only differences being:

- The $\phi$ for such rules have precisely one $\phi(i) = \checkmark$, all the rest not being in its domain.
- $x \in \{\checkmark, \tau\}$.
- If $x = \checkmark$ then $\beta$ is a special value representing $\Omega$, with empty arity.
- If $x = \tau$ then $\phi(i) = \checkmark$ implies $i$ is not in the range of $\psi$.

It should be clear how any scheme of combinator rules can be re-written into a set of tuples of the above form. In fact there is no need of an extended set of operators for giving the semantics of CSP itself: all we need are the operators of CSP, the two half-terminated versions of $\underset{X}{\|}$, and the identity operator *id*. The above machinery allows us to handle much more general languages and semantics.

In the case of $\Sigma$-rules $\rho$ of the form $(\phi, x)$, where the successor term takes the same form as the initial one, we will always have $k(\rho) = 0$ so that $f$ is the empty function/set, and $\psi$ and $\chi$ are the identity functions on their respective domains.

### 3.1 Combinators and supercombinators

As we said in Section 2.3, FDR has for many years run combinators as its main method of state machine evaluation. However it does not work out the operational semantics of a complex process expression (typically, but not exclusively, the application of some parallel, hiding and renaming operators to the perhaps many constituent processes of a network) by applying the combinator rules for the constituent binary and unary operators over processes. Rather it treats the complete construct as a single operator – as we did for entirely different reasons in the re-working of combinators above – and calculates the combinator rules for the entire composition.

These combinator rules for complex constructs have always been called "supercombinators". However it would be wrong to infer that the origin of that name was as extension of the idea of combinators as used in this paper to date. Rather that name came from the analogy with a similar idea from the implementation of functional programming.

So the fact that CSP's non-recursive operators have an operational semantics in terms of (super) combinators has long been "documented" by FDR. The author chose the name "combinators" for this style both because it is extremely natural for what they are, and because they are the natural components from which the long-established idea of supercombinator could be derived. In fact the analogy with functional programming (where there are a few primitive combinators from which all can be derived, but where doing so is more expensive in terms of implementation) turns out to be much better than the author had originally imagined.

More details of the relationship between combinators, supercombinators and FDR's execution model can be found in Chapter 9 of [19].

## 4 CSP simulation without ✓

The fact that all CSP operators are "CSP-like" should not seem very surprising, since obviously we have defined this notion to encompass CSP. The claim which justifies the name of this paper is that all CSP-like operators can be implemented in CSP: there is a CSP expression equivalent to each such operator.

In recent years the author and others have used CSP as a notation into which others can be "compiled", namely translated in such a way that they can be run. Both as part of this work and in response to a variety of other challenges, the author has often been called upon to find ways of expressing in CSP a wide variety of operations – sometimes rather exotic.

The most versatile technique for doing the latter is that of *double renaming*, in which some or all of a process's events are mapped to a pair of separate versions of this event. That then allows us to put our process in parallel with some regulator process which selects which of each of these pairs can occur. This allows us to hide, synchronise etc only those occurrences of a given event that we choose to. The reader will find diverse applications of this idea in [19]. In this section we will use this and other tricks to construct a CSP simulation for every CSP-like operator.

In the rest of this paper we will need to make major extensions to the alphabet of event names that processes are defined over, largely as the sources and targets of these renamings. We will assume that the basic language of processes about which we are reasoning has visible event names drawn from a set $\Sigma_0$, but we will find ourselves extending this significantly to build $\Sigma \supseteq \Sigma_0$. We will assume that all the additional sets added into the alphabet are disjoint from $\Sigma_0$.

We will construct this simulation in stages, developing the techniques we need to simulate different behaviours that CSP-like operators can exhibit. In the first stages we will disregard the possibility of processes terminating $\checkmark$, and so deal only with $\checkmark$-free processes, setting aside $\checkmark$-rules for the time being. For these stages we will be able to build a tighter simulation than when we ultimately allow termination.

**Theorem 1** *Let $\{OP_\lambda \mid \lambda \in \Lambda\}$ be a family of operators over $\checkmark$-free LTSs with CSP-like operational semantics (but no $\checkmark$-rules or rules that generate $\checkmark$), then for each of them $OP_\lambda$ there is a CSP context $C_\lambda[\cdots]$ whose arguments are an $n(\lambda)$-tuple of processes $\mathbf{P}_\lambda = \langle P_1, \ldots, P_{n(\lambda)}\rangle$ and an indexed family $\mathbf{Q}_\lambda = \langle Q_i \mid i \in I(\lambda)\rangle$ of processes such that for all choices of $\mathbf{P}_\lambda$ and $\mathbf{Q}_\lambda$ we have $OP_\lambda(\mathbf{P}_\lambda, \mathbf{Q}_\lambda) = C_\lambda[\mathbf{P}_\lambda, \mathbf{Q}_\lambda]$, equality here meaning strong bisimilarity of transition systems.*

The rest of this section is devoted to constructing these simulations. To aid understanding we will gradually build up the features we allow in the CSP-like operators we consider.

## 4.1 Level 1: all arguments are on and stay on

We will first deal with the case where we neither have to turn processes on nor discard them. In other words we will show how to deal with systems where the set of arguments is constant as it evolves, both the states of these arguments and the operator that is applied to them can vary. Thus the state consists of

- the states of these $n$ fixed arguments,
- the current operator $OP_\lambda$ that is applied to them (with all $OP_\lambda$ having arity $(n, \emptyset)$), and
- a permutation $\pi$ that relates the indices of the fixed argument processes to their positions as arguments of $OP_\lambda$. Because it matches our later needs better, we will record $\pi$ as a function that maps each argument of $OP_\lambda$ to the appropriate index in the original list of arguments.

We can model any operator $OP_\lambda(P_{\pi^{-1}(1)}, \ldots, P_{\pi^{-1}(n)})$ coming from this case as

$$((\|_{r=1}^{n}\ (P_r[\![BR_r]\!], A_r)) \underset{A}{\|} \mathit{Reg}(\lambda, \pi))[\![CR]\!] \setminus H$$

where the $BR_r$ are one-to-many[10] renamings, $A_r$ are process alphabets, the images of $BR_r$, $A = \bigcup_{r=1}^{n} A_r$, $CR$ is a many-to-one renaming and $H$ is a set of actions, all these things being independent of $\lambda$ and $\pi$. $\mathit{Reg}_{\lambda,\pi}$ is a regulator process specific to $\lambda$ and the permutation $\pi$ that gets the various processes to co-operate in the right way. We will call this construction $\mathit{SOP}(\lambda, \pi)[P_1, \ldots, P_n]$ (with $S$ standing for simulated).

The first thing to notice is that irrespective of the values of the various parameters listed above

$$\frac{P_r \xrightarrow{\tau} P_r'}{\mathit{SOP}(\lambda, \pi)[P_1, \ldots, P_r, \ldots, P_n] \xrightarrow{\tau} \mathit{SOP}(\lambda, \pi)[P_1, \ldots, P_r', \ldots P_n]}$$

as a consequence of the operational semantics of the CSP operators used to build $\mathit{SOP}$. This means that the compulsory (and implicit) $\tau$-promotion rules are accurately simulated by construction. This, of course, helps to indicate *why* such rules are necessary.

Our assumptions here mean that every state that $OP_\lambda(P_{\pi^{-1}(1)}, \ldots, P_{\pi^{-1}(n)})$ can reach via the operational semantics will be of the form $OP_\mu(P'_{\pi'^{-1}(1)}, \ldots, P'_{\pi'^{-1}(n)})$ for some $\mu$ and $\pi'$ and states $P_r'$ of $P_r$. What we might therefore expect is that every state of $\mathit{SOP}(\lambda, \pi)[P_1, \ldots, P_n]$ will be of the form $\mathit{SOP}(\mu, \pi')[P_1', \ldots, P_n']$.

Note that the three CSP operators we used in defining $\mathit{SOP}(\lambda, \pi)$ (i.e., renaming, parallel and hiding) all have the property that every state reachable in their operational semantics (ignoring ✓) takes the form of the same

[10]The reader who studies the several definitions of the $BR_i$ given later will realise that they sometimes need the extra generality we allowed ourselves to have renamings where not all process events are in the domain, with others being blocked. $BR$ and $CR$ abbreviate *branching* and *convergent* renaming respectively.

operator (with the same renaming relation, synchronisation set or hidden set) applied to states of the same arguments. It follows from this that every state of $SOP(\lambda, \pi)[P_1, \ldots, P_n]$ has the form

$$((\|_{r=1}^{n} (P''_r[\![BR_r]\!], A_r)) \underset{A}{\|} Reg'))[\![CR]\!]) \setminus H$$

where the $P''_r$ and $Reg'$ are respectively states of the $P_r$ and $Reg(\lambda, \pi)$. It is clear, therefore, that we should aim to have $Reg' = Reg(\mu, \pi')$, $P''_r = P'_r$ and that the same actions should link these states together – something we have already noted for promoted $\tau$s.

As the state evolves, action $a$ from a given process $P_r$ might find itself in any of the following positions: prevented from happening, synchronising with various collections of events from other $P_j$, and mapping to different visible actions as well as sometimes $\tau$. Evidently, in some way, it must be $Reg(\lambda, \pi)$ that decides which of these can happen when (and it is quite possible that there are multiple ways from the same state). This is only possible if the parallel combination $\|_{r=1}^{n} (P_r[\![BR_r]\!], A_r)$ gives $Reg(\lambda, \pi)$ a sufficiently large menu to choose from. So what we will do is have the renamings $BR_r$ create lots of copies of each event that will naturally synchronise in different ways with each other and map to the various target events.

In fact we will rename them to a representation of the overall synchronisation and result that they produce: the combination $(\phi, x)$ of a partial function $\phi : \{1, \ldots, n\} \to \Sigma_0$ and an event in $\Sigma_0 \cup \{tau\}$ where $tau$ is the name of a special visible event in $\Sigma \setminus \Sigma_0$ that represents when the combination synchronises to produce a $\tau$. The renaming $BR_i$ maps the event $a \in \Sigma_0$ to all such pairs $(\phi, x)$ such that $i \in dom(\phi)$ and $\phi(i) = a$. Now let us consider how the process

$$S \ = \|_{r=1}^{n} (P_r[\![BR_r]\!], A_r)$$

behaves. This process cannot perform any of the events $(\emptyset, x)$ (where the function $\phi$ has an empty domain), because none of the $P_i[\![BR_i]\!]$ can. The event $(\{(r, a)\}, x)$ happens whenever $P_r$ performs $a$. $(\{(r, a), (s, b)\}, x)$ happens $(r \neq s)$ represents an $a$ of $P_r$ and $b$ of $P_s$ synchronising to create $x$. We can similarly synchronise any number of the $P_i$ by using $\phi$ with an appropriate domain, including them all synchronising when $\phi$ is a total function. In the result state of these actions it is clear that just those $P_s$ involved in the synchronisation (i.e., $dom(\phi)$) change state, and these $P_s$ change to any state they can on performing the respective $\phi(s)$.

Hopefully it is now clear that, with the exception of rules that depend on no $P_s$ actions at all, the $(\phi, x)$ actions in the combination $S$ give us a way of achieving every possible transition rule that we are allowed under the assumptions of this section other than the $\tau$-promotion ones that are already accounted for. Similarly it should be clear that the final renaming $CR$ will just map each $(\phi, x)$ to $x$, and that $H$ (the set hidden in the definition of $SOP$) is $\{tau\}$.

Just as the form of our simulation *forces* $\tau$ actions to be promoted, the reader should note how it also forces the condition in "CSP-like" that precisely those processes participating in an action change state.

The role of $Reg(\lambda, \pi)$ must therefore be to do the following things

- Allow those events $(\emptyset, x)$ where $OP_\lambda$ performs $x$ with the participation of no arguments. (Note that since none of the arguments will change state, they do not need to be involved.)
- Similarly allow $S$ only to perform those events $(\phi, x)$ where this is an appropriate synchronisation and result for $OP_\lambda[P_{\pi^{-1}(1)}, \ldots, P_{\pi^{-1}(n)}]$.
- Move to a new state that takes account of which operator should be applied to what new permutation of the $P_i$ after such a synchronisation occurs, and indeed after it performs an event with $dom(\phi) = \emptyset$.

The permutation parameter $\pi$ maps the indices of the arguments of $OP_\lambda$ to those of the static group of processes $P_1, \ldots, P_n$ that make up the parallel combination $S$ in our simulation. Thus, when $OP_\lambda$ expects argument $r$ to perform the action $a$, it is the $\pi(r)$th process in the parallel composition that has to perform an $a$. Or in other words, when the fixed processes perform the events represented by the partial function $\phi$, this must correspond to $OP_\lambda$ expecting the functional composition $\phi \circ \pi^{-1}$.

Under the assumptions that we have made in this subsection, all the firing rules of our operator $OP_\lambda$ must take the form $(\phi, x, \mu, \emptyset, \xi, \emptyset)$[11] where $\phi$ are the firing conditions, $x$ is the resulting action, $\mu$ is the index of the resulting operator, and $\xi$ is the permutation mapping the arguments of $OP_\mu$ to those of $OP_\lambda$. It should be clear that this corresponds to the event $(\phi \circ \pi^{-1}, x)$ of $S$, and that the resulting state now has the permutation $\pi \circ \xi$ mapping the arguments of $OP_\mu$ to the participants $P_1, \ldots, P_n$ of $S$. We

[11]The fact that in this subsection we are not considering **off** arguments means that the 4th and 6th components $f$ and $\chi$ of each rule are $\emptyset$.

therefore define[12]

$$Reg(\lambda,\pi) = \Box\{(\phi, v(x)) \to Reg(\mu, \xi \circ \pi) \mid (\phi \circ \pi^{-1}, x, \mu, \emptyset, \xi, \emptyset) \in TR(\lambda)\}$$

where $TR(\lambda)$ are the firing rules of $OP_\lambda$ in the form set out in Section 3, and $v(x) = x$ for all $x \in \Sigma_0$ and $v(\tau) = tau$.

Notice that, unlike $S$, this process *can* perform suitable events in which the partial function $\phi$ is $\emptyset$ (no $P_i$ processes participate in the action). Furthermore such actions do not belong to $A$, the set of actions on which $Reg(\lambda,\pi)$ has to synchronise. It follows from this that the top-level parallel combination in $SOP(\lambda,\pi)$ such actions exactly when $OP_\lambda$ does, independent of what the arguments $P_i$ or cannot do.

Notice also that, for every single firing rule of $OP_\lambda$, including those with $dom(\phi) = \emptyset$ precisely those $P_i$ participating in the corresponding action change state, as required. The following result is now clear because we know that the processes below have exactly the same initial actions, and that each of these leads to one or more pairs of states that are in exactly the same relationship.

**Lemma 1** *For a family of CSP-like operators that are restricted to the form considered in this subsection, the following pair of processes are strongly bisimilar for all choices of $\mu$, $\pi$ and the arguments $P_i$:*

$$SOP(\mu,\pi)[P_{\pi^{-1}(1)}, \ldots, P_{\pi^{-1}(n)}] \qquad \text{and} \qquad OP_\mu(P_1, \ldots, P_n)$$

This lemma then proves Theorem 1 for the case covered in this section.

## 4.2 Step 2: Discarding processes

Having developed a set of techniques for handling the possibilities in Step 1, we will expand them until they cover the full range of CSP-like operators. Essentially there are two further things for us to worry about in the no-$\checkmark$ case: the facts that operators can discard **on** arguments, and that they can make use of **off** arguments by turning them on. In the present subsection we will handle the first of these.

We can therefore assume that every $OP_\lambda$ in a class has $n$ or less **on** arguments (since all those reachable from the consideration of a particular

[12]For this process to serve the exact purpose we intend for it, it must perform *only* the actions implied obviously by this definition. The result would not hold precisely if unfolding this recursion created a $\tau$ action. This is why we have adopted the semantics of recursion that does not create this type of $\tau$.

one cannot have any more arguments than the original). As in the previous section there will be no **off** arguments.

Consider the structure of the process $SOP(\lambda, \pi)[P_1, \ldots, P_n]$ above. It consists of a single parallel/renaming/hiding construction that has a fixed structure throughout its evolution. At all times there will be the $n$ argument processes and the *Reg* running in parallel. We will use a similar structure for this present step, but we will need a way of preventing one or more of the $P_i$ from influencing subsequent behaviour – to the extent that the discarded process's $\tau$s should no longer be allowed to happen.

We choose to do this by putting each $P_r$ in a harness by which it can be turned off and effectively discarded in two different ways. Firstly we allow our process to be *interrupted* by an action in which it does not participate, and secondly we allow events in which it does participate to make it *throw* an exception. We can use a single extra event for the first of these possibilities, but need a second disjoint copy $\Sigma_1 = \{a' \mid a \in \Sigma_0\}$ for the second, since in different circumstances the same event might either discard the process or not. Let $D$ be the renaming that sends $a \in \Sigma_0$ to both $a$ and $a'$. We can define

$$TO(P) = (P[\![D]\!] \,\Theta_{\Sigma_1}\, STOP) \triangle \mathit{off} \to STOP$$

to be the *discard-able* version of $P$ in which the *off* event will move it to $STOP$ unconditionally, while the event $a'$ will turn it off just when $P$ allows the event $a$. Note that in either case the state it moves to after being discarded is $STOP$, which can perform no actions either visible or $\tau$. There is thus a slight difference here between future operational semantic terms and their models: the discarded arguments have disappeared in the former, but in the latter they are still present in the "ghost" form of $STOP$ – more turned off than discarded.

Suppose we are given the firing rule $\rho = (\phi, x, \mu, \emptyset, \psi, \emptyset)$ of a general operator $OP_\lambda$ with arity $n$, satisfying the assumptions of this middle step. Then we can compute $Discard(\phi)$ from this, namely the set of process indices that are discarded by it firing, either by interrupt or throw. For the first sort we need the firing of this rule to send an *off* signal, and for the second we need the rule to make $TO(P_r)$ perform $\phi(r)'$ rather than $\phi(r)$.

We will therefore extend the events of our simulation to take the form $(\phi, x, B)$ where $B$ is the set of arguments discarded by the corresponding rule $\rho$: $B = Discard(\rho)$. The renamings $BR_r$ need to be extended to accommodate this:

$$\begin{array}{rcl} BR_r & = & \{(a, (\phi, x, B)) \mid \phi(r) = a \wedge r \notin B\} \\ & & \cup \{(a', (\phi, x, B)) \mid \phi(r) = a \wedge r \in B\} \\ & & \cup \{(\mathit{off}, (\phi, x, B)) \mid r \notin \mathit{dom}(\phi) \wedge r \in B\} \end{array}$$

The first line here covers the ways $P_r$ can proceed normally without being discarded. The second line covers the case where $P_r$ participates in the event that discards it, and the third when it is discarded by things external to it.

We will create a simulation with the same overall structure as that in the previous subsection, except that the $P_r$ are replaced by $TO(P_r)$. The alphabet $A_r$ of $TO(P_r)$ is the range of this expanded renaming, and $CR$ will now map triples of the form $(\phi, x, B)$ to $x$. We note that triples of the form $(\phi, x, B)$ where no events participate in the action can still have $B \neq \emptyset$ and discard processes.

In $Reg(\mu, \psi)$, $\psi$ is not now a permutation, but an injective function from $\{1, \ldots, n(\mu)\}$ to $\{1, \ldots, n\}$, with $n \geq n(\mu)$ being the uniform bound on arities discussed above. It tells us which of the original $n$ argument processes is playing the role of each argument of $OP_\mu$. All the original processes not in the range of $\psi$ will have been discarded by previous actions when this state is reached and therefore become $STOP$ in the simulation.

We can therefore define

$$\begin{array}{rcl} Reg(\lambda, \psi) & = & \Box\{(\phi \circ \psi^{-1}, v(x), \psi(Discard(\rho)))) \rightarrow Reg(\mu, \psi \circ \psi') \mid \\ & & \rho = (\phi, x, \mu, \emptyset, \psi', \emptyset) \in TR(\lambda)\} \end{array}$$

For classes of operators covered by this the assumptions of this subsection, the model of any operator $OP_\lambda(\mathbf{P})$ ($\mathbf{P} = \langle P_1, \ldots, P_n \rangle$) with $n(\lambda) = n$ is now $SOP(n, \lambda, id)[\mathbf{P}]$, where for general $n \geq n(\mu)$, $\mu$, $n(\mu)$-tuples of processes $\mathbf{V}$ and $\psi$ we define

$$SOP(n, \mu, \psi)[\mathbf{V}] = ((\|_{r=1}^{n} (\mathcal{Q}(\mathbf{V}, \psi, r)[\![BR_r]\!], A_r)) \underset{A}{\|} Reg(\mu, \psi))[\![CR]\!] \setminus H$$

where $\mathcal{Q}(\mathbf{V}, \psi, r) = TO(V_{\psi^{-1}(r)})$ if $r \in range(\psi)$ and $STOP$ otherwise. The first argument ($n$) of $SOP$ is needed to establish how many of the $STOP$ ghosts of already-discarded processes are present in the simulation it creates.

The accuracy of our model is expressed in the following lemma.

**Lemma 2** *Suppose we have a family of operators $\{OP_\lambda \mid \lambda \in \Lambda\}$ satisfying the assumptions of this subsection and that $n(\lambda) \leq n$ for all $\lambda$. Suppose* *that $\lambda \in \Lambda$ and that $\psi : \{1, \ldots, n(\lambda)\} \to \{1, \ldots, n\}$ is an injective function. Then the following pair of processes are strongly bisimilar for all choices of operands $\mathbf{P} = \langle P_1, \ldots, P_{n(\lambda)} \rangle$:*

$$SOP(n, \lambda, \psi)[\mathbf{P}] \qquad \text{and} \qquad OP_\lambda(\mathbf{P})$$

The proof of this is to show that each action of one of these two processes is mirrored by one of the other.

- We note that either process can perform a promoted $\tau$ action when any one of $P_1, \ldots, P_{n(\lambda)}$ does. The rest of the parallel processes in $SOP(n, \mu, \psi)[\mathbf{P}]$ are *STOP* or a state of *Reg*, so none of these can perform a $\tau$ to promote.

- The argument relating to actions based on transition rules is essentially the same as in Step 1.

- Note that in either way of discarding a process, it is turned off (to *STOP*) by the specific action that causes this rather than an additional action. There are thus no intermediate extra states created by our discarding mechanism.

- Since every action discards precisely those arguments that are no longer required by the successor operator, we maintain the invariant that the processes that have not been discarded are exactly $range(\psi)$, so that the processes running in the simulation are always $TO(V_j)$ ($j \in \{1, \ldots, n(\mu)\}$) for whatever processes $\mathbf{V}$ are the arguments of the current $OP_\mu$.

## 4.3 Turning processes on

Our full definition of a CSP-like operational semantics implies that argument processes processes can be held in reserve and turned on later, and that constant processes can be introduced in the places of **on** arguments for $OP_\mu$ that are reached during a run. These processes, unlike **on** ones, may be copied, but only before they have become **on**. This really presents two separate challenges in extending our simulation. Of them the second (copying) is harder to deal with, not least because we can no longer use the static pattern of $n + 1$ processes that has served us until now.

If we banned the copying of **off** arguments, there were only finitely many of these, and we forbade the introduction of constant processes then there would be little problem. We could then again assume that we had a fixed-size finite population of process arguments: some of them **on** and running, some of them already discarded, and some **off** ones yet to be turned on. We could include each of the third class as a component in our parallel composition as $on \to TO(Q)$. Notice that as long as they remain in this form they are unable to perform any action other than *on*, and so in particular will not have any $\tau$ actions to be promoted automatically through the structures of *SOP*.

The process of turning the **off** arguments **on** would then be exactly analogous to the way we discard a process using *off* in the previous section, namely when the process itself did not participate in the discarding action. In other words we would include an extra set of process labels in the main action model of *SOP* so they become $(\phi, x, B, C)$, where $C$ is the set of components to be turned on (disjoint from $dom(\phi)$ and $B$), and rename the *on* of process $r$ to those events where $r \in C$.

But we have set ourselves a sterner challenge, namely managing a parallel composition that is dynamic in length. We should note, however, that these parallel compositions never need to become infinite since no operator has an infinite number of **on** arguments.

Dynamic parallel compositions can easily be created in CSP by recursing through parallel operators. This has been done since the earliest days of CSP [4], frequently using the chaining[13] $\gg$ and enslavement $/\!/$ operators. In order for it to work (in the sense of not simply creating an undefined or divergent process), it must be impossible for an infinite chain of unfoldings of these parallel operators to occur without an infinite sequence of visible events also occurring. The issues surrounding this "guardedness" with chaining and enslavement were extensively studied in [13].

Thus in our case we should expect to start with a finite parallel composition including a single process containing both all of the **off** arguments and the constant processes that operator definitions use. It should be capable of spawning off any combination of these processes that might be turned on by a single action, so that these get added into the pool of processes simulating **on** arguments. It should remain in the parallel composition after spawning off such combinations, since it might have to do similar things on later actions.

The reader will recall that the parallel composition inside our existing *SOP* processes already has a very tangled web of potential synchronisations. The situation can only get worse when we envisage an arbitrarily large collection of processes that might have to synchronise in any combina-

[13]See Section 8, where we devise a variant chaining operator, for examples.

tion. Fortunately, however, our previous choice of finite partial functions $\phi$ still works. Each $\Sigma_0$ will now be renamed to an infinite set of actions even when $\Sigma_0$ itself is finite, since the $\phi$s may now have any finite subset of $\mathbb{N}$ as their domains.

Taking account of the need for a process that can spawn off further copies of the **off** arguments $\mathbf{Q}$, the inner process of our simulation will now take the form:

$$S = (\|_{r=1}^{m} (V_r[\![BR_r]\!], A_r)) \ {}_{A_m^+}\|_{E_m} \ \mathit{Resources}(I, \mathbf{Q}, m)$$

where the $V_r$ are states relating to $TO(P_r)$ where $P_r$ is either some initial argument or a process $Q(i)$ that has been turned on, $A_r^+ = \bigcup\{A_i \mid i \leq r\}$ and $E$ is the set of all tuples representing transitions. $\mathit{Resources}(I, \mathbf{Q}, r)$ is a process that spawns off new processes from the family $\mathbf{Q}$ indexed by $I$, giving each process it spawns off a unique index in the overall composition that increases. (These indices will start from $n + 1$, where $n$ is the number of initially **on** arguments, and increase.)

We will use events with five components $(\phi, x, B, m, f)$ where $\phi$ is a finite partial function from $\{1, \ldots, m\}$ to $\Sigma_0$, $x \in \Sigma_0 \cup \{tau\}$, $B$ is a finite subset of $\{1, \ldots, m\}$ representing the processes to be discarded, $m$ is (as above) the number of parallel components already created and $f$ is a finite partial function with domain in $\{1, \ldots, k\}$ for some $k \geq 0$ to $I$, the indexing set (which we assume is extended to accommodate an index for every constant process that operators introduce).

The role of $f$ is to assign indexed processes to whichever new parallel slots this operator creates, just as it did in our re-casting of combinator rules.

The process $\mathit{Resources}(I, \mathbf{Q}, m)$, which plays the role of all the processes that have not yet been turned on and where $m$ processes already exist, may be defined as follows

$$\begin{array}{l} \mathit{Resources}(I, \mathbf{Q}, m) \ = \\ \Box\{(\phi, x, B, m, f) \to \\ \quad \|_{r=m+1}^{m+|f|} (TO(Q(f(r-m)))[\![BR_r]\!], A_r)) \ {}_{A^{\#}_{m,m+|f|}}\|_{E} \ \mathit{Resources}(I, \mathbf{Q}, m+ \mid f \mid) \\ \quad \mid (\phi, x, B, m', f) \in E \wedge m = m'\} \end{array}$$

where $A^{\#}_{k,l} = \bigcup_{r=k+1}^{l} A_r$ and $E$ is, as defined above, the set of all the 5-tuples representing events.

In other words, it listens to the last two components in the next event and splits itself into a parallel composition of newly-on processes that can join the existing flock and a copy of itself adjusted to start off the next group with the correct indexes.

Notice here that a single event has the power to turn on an arbitrarily large finite collection of processes, which may include many copies of the same $Q(i)$. Notice also that though $\mathit{Resources}(\cdot)$ has all events in its alphabet, it allows all events that correspond to the current size of the system, and so does not restrict what transitions are possible in the present state. Its only role is to reconfigure the system for the next

The initial state of the as-yet unconstrained family of processes is thus

$$(\|_{r=1}^{n} (TO(P_r)[\![BR_r]\!], A_r)) \;{}_{A_n^+}\|_E\; \mathit{Resources}(I, \mathbf{Q}, n)$$

with the renamings $BR_r$ being extended to

$$\begin{array}{rcl} BR_r & = & \{(a, (\phi, x, B, f, m)) \mid \phi(r) = a \wedge r \notin B\} \\ & & \cup\, \{(a', (\phi, x, B, f, m)) \mid \phi(r) = a \wedge r \in B\} \\ & & \cup\, \{(\mathit{off}, (\phi, x, B, f, m)) \mid r \notin \mathit{dom}(\phi) \wedge r \in B\} \end{array}$$

and $A_r$ again being the image of $BR_r$. In understanding how this system evolves it is important to remember that the alphabetised parallel operator ${}_X\|_Y$ of CSP is both commutative and associative under natural actions on the alphabets. So it does not matter what order a list of alphabetised processes appear in, or the structure of bracketing. In particular, the processes "spun off" by $\mathit{Resources}(I, \mathbf{Q}, n)$ have the same effect as though they were combined directly with the already turned-on components: given that all $A_m^*$ and $A_{i,j}^*$ are subsets of $E$

$$(P \;{}_{A_m^+}\|_E\; (Q \;{}_{A_{m,m+r}^*}\|_E\; R) = (P \;{}_{A_m^+}\|_{A_{m,m+r}^*}) \;{}_{A_{m+r}^+}\|_E\; R$$

We need to extend the controlling process $\mathit{Reg}$ to allow for the dynamic network and the need to turn processes on as well as discard them. Its parameters are now the current operator $\lambda$, an integer $m$ saying how many processes have already been started up, an injective function $\psi$ that maps $\{1, \ldots, n(\lambda)\}$ to $\{1, \ldots, m\}$ which establishes which of the $m$ processes each of the **on** arguments of $OP_\lambda$ is, and a function $\chi : I(\lambda) \to I(\lambda_0)$ which maps each index of an **off** argument of $OP_\lambda$ to an index of one of the original **off** arguments: **off** arguments, by their nature, have not made any progress since the start. A consequence of our definition CSP-like operators and the fact that we have accommodated all introduced constant processes as **off** arguments is that all the **off** arguments of a successor operator are **off** arguments of the original. $\lambda_0$ is assumed to be the index of the operator at the root of the derivation tree, whose **off** operands necessarily contain those of all other operators derived from it.

$$\begin{array}{rcl} \mathit{Reg}(\lambda, m, \psi, \chi) & = & \Box\{(\phi \circ \psi^{-1}, x, \psi(\mathit{Discard}(\rho)), \chi \circ f, m) \rightarrow \\ & & \qquad \mathit{Reg}(\mu, m+ \mid f \mid, (\psi \cup \mathit{id}_{\{m+1,\ldots,m+|f|\}}) \circ \psi', \chi \circ \chi') \mid \\ & & \rho = (\phi, x, \mu, \psi', \chi', f) \in \mathit{TR}(\lambda)\} \end{array}$$

The complicated construction $(\psi \cup \{m+1, \ldots, m+ \mid f \mid\}) \circ \psi'$ of the new function mapping the **on** arguments of $OP_\mu$ to indices in the simulation says that

- If a component has been freshly created by this transition (i.e. the result of applying $\psi'$ to it gives a result greater than $m$) then it is mapped directly to the result of $\psi'$. Here $\mathit{id}_A$ is just the identity function on the set $A$.

- If a component was already in existence before the rule fires (i.e. $\psi'$ maps it to an index no greater than $m$) then we need to compose $\psi'$ with the function $\psi$ that determines the pre-rule **on** arguments.

Now, given a finite list $\mathbf{P} = \langle P_1, \ldots, P_{n(\mu)}\rangle$ of **on** arguments, any $m \geq n(\mu)$, an indexed (by $I_0$) family of **off** arguments $\mathbf{Q}$, an injective function $\psi$ from $\{1, \ldots, n(\mu)\}$ to $\{1, \ldots, m\}$ and a function $\chi$ from $I(\mu)$ to $I_0$, we can define

$$\begin{array}{l} \mathit{SOP}(m, \mu, \psi, \chi)[\mathbf{P}, \mathbf{Q}] = \\ (((\|_{r=1}^{m} (\mathcal{Q}(\mathbf{P}, \psi, r)[\![BR_r]\!], A_r)) \;{}_{A_m^+}\|_{A_m^-}\; \mathit{Resources}(I, \mathbf{Q}, m)) \\ \underset{E}{\|}\; \mathit{Reg}(\mu, m, \psi, \chi))[\![CR]\!] \setminus H \end{array}$$

where $A = \bigcup\{A_r \mid r \in \{1, 2, 3, \ldots\}\}$.

This system is intended to behave in the same way as the corresponding processes in the previous two sections in respect of the types of behaviour dealt with there. However this one has the capability of expanding so that its parallel composition is of arbitrary finite size. Note that a given action $(\phi, x, B, m, f)$ can synchronise an arbitrary collection of the running processes, close down any group of them, and start up $m$ new ones all at once! Since all this can happen in a single action in our definition of a CSP-like operational semantics this is, of course, vital. The lemma that corresponds to our earlier one is the following.

**Lemma 3** *Suppose we have, in our ✓-free case, a family of CSP-like operators $\{OP_\lambda \mid \lambda \in \Lambda\}$, and that $\lambda \in \Lambda$, $m \geq n(\lambda)$, that $\psi : \{1, \ldots, n(\lambda)\} \rightarrow$*

$\{1, \ldots, m\}$ *is an injective function and* $\chi : I(\lambda) \to I_0$*. Then the following pair of processes are strongly bisimilar for all* $\mathbf{P} = \langle P_1, \ldots, P_{n(\mu)} \rangle$ *and families of processes* $\mathbf{Q}$ *indexed by* $I_0$*:*

$$\mathit{SOP}(m, \mu, \psi, \chi)[\mathbf{P}, \mathbf{Q}] \qquad \text{and} \qquad \mathit{OP}_\lambda(\mathbf{P}, \mathbf{Q} \circ \chi)$$

Just as with our previous lemmas, this one is constructed so that every state a process of the above type goes through is of the same type. Furthermore the correspondences already established for the earlier lemmas, and our constructions allowing an action to turn processes from $\mathbf{Q}$ on in this section, mean that the actions of the two sides are in exact correspondence.

As in the previous section, as the *SOP* simulations progress they are left with a *STOP* process for every argument process that has been discarded. The difference between $m$ and $n(\mu)$ will always be exactly the number of such *STOP*s.

Setting $\mu = \lambda$, $m = n(\mu)$, $I_0 = I(\lambda)$, with $\psi$ and $\chi$ both being the identity function gives us Theorem 1: $C_\lambda = \mathit{SOP}(n(\lambda), id_{\{1\ldots,n(\lambda)\}}, id_{I(\lambda)})$.

The reader might note that the simulation created for the *Farm*$(P)$ operator (in which an **off** argument is run in parallel with itself an arbitrary number of times) is, at least in the *Resources* process, rather similar to the CSP definition *Farm* itself given earlier.

## 5 Simulation with ✓

The CSP language restricts what can be done with ✓, like $\tau$ is not referred to directly in programs, so it cannot be renamed, hidden using $\setminus X$, included optionally in synchronisation sets or introduced via prefixing: it only occurs through the execution of *SKIP*. There is good reason for most of these restrictions if we want to respect the idea that ✓ is an observable but uncontrollable signal: the reader will have seen the effect of these special properties on our definition of the form of a combinator semantics.

This means that the rather amazing coding tricks used earlier, available through the use of renaming and synchronisation with a specially crafted process, are not available directly on the action ✓. It follows therefore that a regulator process cannot be guided directly by the termination event of one of the argument processes. Equally, if the complete simulation *SOP* is to terminate, then all the components must brought to a state where they can terminate, though they do not have ✓ immediately available.

Consider how our simulation, at the level explained in Section 4.2, would work for the combinator operational semantics of the operators $P \,\square\, Q$ and

$P \triangle Q$. In each case, either argument terminating leads to the construct terminating, even when the other argument does not. $TO(P)$ or $TO(Q)$ would, in these simulations, still be running in parallel with a regulator. One's initial reaction to this might be to give $TO(P)$ the additional ability to terminate via ✓ even when $P$ does not, so it can "match" the termination of $Q$. We could do this by replacing it with $TO(P) \triangle \mathit{SKIP}$. But for various reasons this does not work, not least because such a ✓ would not be controllable by the *Reg* process. We therefore have to make the regulator instruction to $TO(P)$ to terminate be guarded by an ordinary event.

Thus, if we were to run

$$\begin{array}{l} TO((P;\ \mathit{term}.1 \to \mathit{SKIP}) \triangle (\mathit{quit}.1 \to \mathit{SKIP})) \quad \text{and} \\ TO((Q;\ \mathit{term}.2 \to \mathit{SKIP}) \triangle (\mathit{quit}.2 \to \mathit{SKIP})) \end{array}$$

in parallel with *Reg*, then $P$ terminating sends the *term*.1 signal to *Reg*. We can arrange that *Reg*, on receiving this, then communicates *quit*.2 that forces the other argument to terminate, and then terminates itself. All three parallel components would then terminate, so that the externally visible behaviour of the simulation would be exactly what is needed.

Analysing what happens here: the ✓ event of $P$ is hidden by sequential composition and followed by the event *term*.1 (which would be hidden, like the other newly-introduced ones, at the outside of the simulation, therefore becoming $\tau$). This event would trigger *Reg* to send the *quit*.2 event – also an external $\tau$ – followed by a chain of ✓s as the three parallel arguments including *Reg* (hidden by $\|$ to become $\tau$s) terminate, triggering in turn the termination of the two $\|$ operators involved. [In fact other interleavings of these events are also possible.] So what, in $P \square Q$, would have been a single ✓ event, has become ✓ preceded by no less than 7 $\tau$s at the top level. The behaviour is in fact equivalent in every CSP model, but we have certainly lost the strong bisimulation achieved by our previous simulations.

We can improve this a little in the sense of reducing the number of extra $\tau$s by using the throw operator. There are two options for this. In the first of these we replace the $TO((P;\ \mathit{term}.i \to \mathit{SKIP}) \triangle (\mathit{quit}.i \to \mathit{SKIP}))$ processes by $TO(P;\ \mathit{term}.i \to \mathit{STOP})$. *Reg* then communicates *exit* (not synchronised with any other process) when it wants the entire simulation to terminate, and the result placed in the context

$$(S\, \Theta_{\mathit{exit}}\, \mathit{SKIP}) \setminus \{\mid \mathit{term}, \mathit{exit} \mid\}$$

This removes the extra $\tau$s caused by the distributed termination mechanism in our approach above. It also simplifies the components (no longer requiring *quit.i*) and *Reg* (since in general *Reg* would, in the earlier approach, have to issue events that synchronise with several *quit.i* at the same time).

We also need to consider the permitted case where a $\Sigma$ rule creates a $\checkmark$. Just as we introduced a new event *tau* as the image of rules with output event $\tau$, so we use *exit* as the image of those rules creating $\checkmark$.

It is possible to capture exactly what this mechanism achieves: an extra $\tau$ is introduced before any component process terminates, and a further extra one (i.e. the hidden *exit*) before the whole terminates. In addition, when (as in $P \;\Box\; Q$) a $\checkmark$ of an argument is transmitted directly to the outside, we get an extra $\tau$ caused by hiding the *term.i* that is not present in the original operational semantics. [The $\tau$ from hiding *term.i* is present when the combinator semantics turns a $\checkmark$ of argument **i** into $\tau$ – namely this very $\tau$.]

This lack of consistency can be eliminated if we replace $TO(P;\ term.i \rightarrow STOP)$ by $TO(P;\ (iterm.i \rightarrow STOP \,\Box\, xterm.i \rightarrow STOP))$. In other words whenever an argument terminates, it gives *Reg* the choice of it communicating one of two events: every state of *Reg* will allow at least one of these two events for every **on** arguments (usually exactly one). $\checkmark$ rules of the form $((\ldots,\checkmark,\ldots),\checkmark)$ correspond to the event *xterm.i* (i.e. externally visible termination), and ones of the form $(\ldots,\checkmark,\ldots),\tau,Q)$ correspond to the event *iterm.i* (internalised termination). The external context then becomes

$$S\,\Theta_{\{|exit,xterm|\}}\,SKIP) \setminus \{|\ iterm, xterm, exit\ |\}$$

Now, the event indicating that $P_i$ has terminated becomes *xterm.i* if the combinator semantics says that a $\checkmark$ of $P_i$ is promoted to $\checkmark$. This then triggers $\Theta_{\{|exit,xterm|\}}$ and brings about overall termination.

Imagine running arguments of the form $P;\ SKIP$ in the original context (e.g. $(P;\ SKIP) \;\Box\; (Q;\ SKIP)$). This introduces exactly the same extra $\tau$ before one of the arguments terminates as in our final simulation. The only difference is that this can *then* communicate $\checkmark$ to the world when our simulation performs hidden *xterm.i* which in turn then leads to $\checkmark$. In cases analogous to $P \underset{X}{\|} Q$ where the termination signals of the arguments become *iterm.i* events, the execution of our simulation exactly follows $(P;\ SKIP) \underset{X}{\|} (Q;\ SKIP)$ up to the point when the final $\checkmark$ occurs, for this is replaced by a hidden *exit* followed inevitably by the tick of the *SKIP* that follows the $\Theta$. Note that either case where $\Theta_{\{|exit,xterm|\}}$ is triggered, all the component processes are closed down immediately by the exception and are no longer even able to perform $\tau$. It follows that the overall simulations in our two cases are strongly bisimilar to

$$((P;\ \mathit{SKIP}) \,\square\, (Q;\ \mathit{SKIP}));\ \mathit{SKIP} \quad \text{and} \quad ((P;\ \mathit{SKIP}) \underset{X}{\parallel} (Q;\ \mathit{SKIP}));\ \mathit{SKIP}$$

So we have the following generalisation of Theorem 1.

**Theorem 2** *Let $\{OP_\lambda \mid \lambda \in \Lambda\}$ be a family of operators over LTSs with CSP-like operational semantics, then for each of them $OP_\lambda$ there is a CSP context $C_\lambda[\cdots]$ whose arguments are an $n(\lambda)$-tuple of processes $\mathbf{P} = \langle P_1, \ldots, P_{n(\lambda)}\rangle$ and an indexed family $\mathbf{Q} = \langle Q_i \mid i \in I(\lambda)\rangle$ of processes such that for all choices of $\mathbf{P}$ and $\mathbf{Q}$ we have $OP_\lambda(\mathbf{P}^{SKIP}, \mathbf{Q}^{SKIP});\ \mathit{SKIP} = C_\lambda[\mathbf{P}, \mathbf{Q}]$, equality here meaning strong bisimilarity of transition systems. Here, the $i$th component of $\mathbf{P}^{SKIP}$ is $P_i;\ \mathit{SKIP}$.*

This is a generalisation because under the restrictions of Theorem 1 we would have $P_i;\ \mathit{SKIP}$ bisimilar to $P_i$, $Q_i;\ \mathit{SKIP}$ bisimilar to $Q_i$ and $OP(\mathbf{P}, \mathbf{Q});\ \mathit{SKIP}$ bisimilar to $OP(\mathbf{P}, \mathbf{Q})$.

Certainly we would expect the terms

$$OP_\lambda(\mathbf{P}^{SKIP}, \mathbf{Q}^{SKIP});\ \mathit{SKIP} \quad \text{and} \quad OP_\lambda(\mathbf{P};\ \mathit{SKIP}, \mathbf{Q};\ \mathit{SKIP});\ \mathit{SKIP}$$

to be equivalent thanks to one of the basic laws of CSP: $P;\ \mathit{SKIP} = P$. So the equivalence proved in the above result is exactly in line with what we would expect.

If we knew that $OP_\lambda$ was a well-defined operator over all CSP models then this law would prove that these two terms (considered as LTSs) are indeed equivalent over such models. But we do not quite know this yet.

We *do* know it for operators coming under Theorem 1 over the ✓-free versions of the CSP models, since if, the operator $OP_\lambda$ were not well defined over such a model, there would be arguments $(\mathbf{P}, \mathbf{Q})$ and $(\mathbf{P}', \mathbf{Q}')$ equivalent over the model where $OP_\lambda(\mathbf{P}, \mathbf{Q})$ and $OP_\lambda(\mathbf{P}', \mathbf{Q}')$ are not. But we know that these two terms are strongly bisimilar, and therefore model-equivalent, to their simulations, which certainly are model-equivalent by the well-definedness of the operators the simulations are constructed from.

Where we are lacking in the case with ✓s is the knowledge that $T_1 = OP_\lambda(\mathbf{P}, \mathbf{Q})$ and $OP_\lambda(\mathbf{P}^{SKIP}, \mathbf{Q}^{SKIP});\ \mathit{SKIP}$ are equivalent in every CSP model, which is the same, whatever value $T_2 = OP_\lambda(\mathbf{P}^{SKIP}, \mathbf{Q}^{SKIP})$ has, as asserting that $T_1$ and $T_2$ are equivalent in CSP models since in all such models $T_2;\ \mathit{SKIP}$ is equivalent to $T_2$.

**Lemma 4** *Suppose $OP_\lambda$ is a CSP-like operator defined by combinator operational semantics, then in every CSP model $OP_\lambda(\mathbf{P}, \mathbf{Q})$ and $OP_\lambda(\mathbf{P}^{SKIP}, \mathbf{Q}^{SKIP})$ are equivalent.*

PROOF As set out in [19], the value of a process in every CSP model is obtained from the set of linear observations that can be made of processes. These consist of finite and infinite sequences of visible events and acceptance sets (sets offered by the process in stable, i.e. $\tau$ and $\checkmark$-free, states), perhaps ended by a marker for divergence. There can be at most one acceptance set between consecutive visible events, and none before a $\checkmark$, which is necessarily final. The form of the recorded behaviours is simplified if we use a special symbol $\bullet$ to represent the absence of the observation of a stable acceptance between two visible events: we strictly alternate acceptances $A$ (with the option of $\bullet$) and the events. All observations therefore take one of the following forms, where $a_i \in \Sigma$ and $A_i$ is either $\bullet$ or a subset of $\Sigma$.

1. $\langle A_0, a_1, A_2, \ldots, A_{n-1}, a_n, A_n \rangle$ finite unterminated observation.
2. $\langle A_0, a_1, A_2, \ldots, A_{n-1}, a_n, \bullet, \checkmark \rangle$ observation leading to process termination.
3. $\langle A_0, a_1, A_2, \ldots \rangle$ observation with an infinite trace.
4. $\langle A_0, a_1, A_2, \ldots, A_{n-1}, a_n, \Uparrow \rangle$ observation ending in divergence.

Note that where $A_{r-1} \neq \bullet$, necessarily $a_r \in A_{r-1}$.

We can prove that two processes are equivalent in any CSP model by proving that they have the same such observations.

It is clear that the only effects of replacing an argument $P$ by $P;\ SKIP$ in an operator with combinator operational semantics come in states where $P$ is both **on** and in one of its own states that can communicate $\checkmark$. Aside from that, $P$ and $P;\ SKIP$ have exactly the same actions that influence the overall semantics, and proceed themselves exactly in step.

Let $C[\cdot]$ be the context in which these arguments sit, so we are considering the relative behaviours of $C[P]$ and $C[P;\ SKIP]$.

We can identify the states of the operational semantics of $C[P]$ with *configurations*: the pieces of syntax that the operational semantics has generated by this point in the derivation.

Configurations of $C[P]$ in which $P$ can perform $\checkmark$ are replaced by exactly two of the version with $P;\ SKIP$: the original one where the argument can now perform $\tau$ rather than $\checkmark$, and the one where it can now *only* perform $\checkmark$. Note that no such configuration (of the original system or transformed) can be stable since they can always perform $\checkmark$ or $\tau$.

It should be clear that for every trajectory of $C[P]$ (i.e. a sequence of its configurations starting from $C[P]$ and the top-level events that relate them) there is one of $C[P;\ SKIP]$ which is identical except that

- If the configuration of $C[P]$ contains a state $P'$ of $P$, then this is replaced by $P'$; *SKIP*.
- Any of the trajectory's transitions $Q \xrightarrow{x} Q'$ (necessarily with $x \in \{\tau, \checkmark\}$) in which a copy of the argument $P$ performs a $\checkmark$ is replaced by two: the first is a $\tau$ in which the configuration is unchanged except that this particular copy of $P$ becomes *SKIP*. In the second the *SKIP* performs its $\checkmark$ resulting in the overall $x$.

Note that if $P$ is an **off** argument then $C[P]$ might run many copies of it or none at all: it follows that more than one transition of our trajectory might need the above transformation as various copies of $P$ terminate.

It should similarly be clear that any trajectory of $C[P;\ \mathit{SKIP}]$ may be reduced to one of $P$ by the following transformation.

- All sub-terms of the form $P'$; *SKIP* become $P'$, for $P'$ a state of $P$.
- All sub-terms of the form *SKIP*, where this *SKIP* has arisen because of a transition $P' \xrightarrow{\checkmark} \cdot$ creating the $\tau$ of $P'$; *SKIP* that moves this process to *SKIP*, is replaced by $P'$. [Note that in many cases $P'$ will itself be *SKIP*, but could also, for example, be $\mathit{SKIP} \mathrel{\square} Q$ for some $Q$.]
- One or more $\tau$ actions in the trajectory may have come from the promotion of the $\tau$ generated by $P';\ \mathit{SKIP} \xrightarrow{\tau} \mathit{SKIP}$ for a state $P'$ of $P$ such that $P' \xrightarrow{\checkmark} \cdot$. Note that the first two of our transformations make the "before" and "after" states of this transition the same. We simply delete this transition: it is the extra one created by putting an extra $\tau$ before the component's $\checkmark$. You should notice that this will never remove an infinite consecutive series of transitions since for this to happen there would have to be infinitely many **on** copies of $P$ active at the same time, and this is impossible.

In each direction of this transformation the set of linear observations that can be made, of the forms above, are identical. Certainly the series of visible events is unaltered, and no stable state has its initial set of events (i.e. acceptance set) altered. (Recall that no state with a $\checkmark$ or $\tau$ available is stable.) Finally, it is clear that any trajectory that ends in divergence is transformed to another that ends in divergence in either direction.

We gave the above transformations based on a single argument $P$, but the same would have worked for an arbitrary collection. This completes the proof of the lemma. ∎

We can deduce that the value, abstracted from the operational LTS to any CSP model, of our simulation, corresponds precisely to the similarly obtained value of the process being simulated. This implies the following result, which we have already demonstrated for the $\checkmark$-free case.

**Theorem 3** *If $OP_\lambda(\mathbf{P}, \mathbf{Q})$ is a CSP-like operator then, for any CSP model $\mathcal{M}$, the $\mathcal{M}$-value of $OP_\lambda$ applied to its arguments depends only on the $\mathcal{M}$-values of its arguments.*

It is of course this result which, more than any other, justifies the name "CSP-like".

## 6 Theoretical ramifications

The fact that programs built from CSP-like operators have exact or near-exact simulations using combinations of CSP operators tells us that they share many long-established properties of CSP programs. The first is that, as shown above, any CSP operator has a well-defined and operationally congruent semantics over every model for CSP, such as traces $\mathcal{T}$ and failures-divergences $\mathcal{N}$, belonging to the hierarchy of behaviourally-based models set out in [19].

This semantics is simply that implied by the simulation: we can treat every CSP-like operator as derived in the same sense that chaining and its generalisation, the link parallel operator $P[a \leftrightarrow b]Q$, are derived from parallel, renaming and hiding. (See Chapter 5 of [19].)

In formulating the following result we need to bear in mind that each of these models has its own fixed-point theory that is used to find the semantics of recursive processes, and that adding a number of extra operators which are all CSP definable makes no difference to the correctness (i.e. operational congruence) of a given fixed-point theory. Three different such theories are used in [19]: one for models like $\mathcal{T}$ based only on behaviours that can be observed in a finite time, one for those like $\mathcal{N}$ that treat every divergent process as equivalent to the least refined one (i.e. *divergence strict* models) and a more difficult one for models that have infinite behaviours without divergence strictness.

**Theorem 4** *Suppose we are given an alphabet $\Sigma_0$ of visible actions, a set of constant processes represented by LTSs over $\Sigma_0 \cup \{\tau\}$, a collection of CSP-like operators over such LTSs and recursion (whose operational semantics includes the additional $\tau$ to avoid undefined terms). Then every semantic* *model for CSP also gives a model for the resulting language $\mathcal{L}$ whose semantics for recursion takes the same shape as CSP's over the same model. Furthermore, terms of $\mathcal{L}$ are monotonic with respect to the normal CSP definition of refinement over such models.*

This follows straightforwardly from the fact that CSP-like operators are equivalent to their CSP simulations in every model, and the corresponding facts for CSP.

This is a result that will have many consequences for any such language $\mathcal{L}$. For example it means that, whether implemented via our CSP simulation *SOP* or otherwise, we can confidently apply any function of FDR, including its sometimes model-specific compression operators, to $\mathcal{L}$ programs in the same way as we do to CSP. Furthermore, for the purpose of analysis in CSP models, all the operators of CSP can be added to $\mathcal{L}$. We could choose to draw up the specification side $P$ of a refinement check $P \sqsubseteq Q$ in $\mathcal{L}$, and the other in CSP, or *vice-versa.*

We will discuss the issues relating to the implementation of CSP-like operators in CSP$_M$ and directly in FDR in Section 7.

## 6.1 Distributivity

CSP-like operators inherit other properties from those of the CSP operators used to define the simulation. Recall that all CSP operators other than recursion are *distributive*: for any non-empty set $S$ of processes and operator $op$, if all the operands of $op$ other than one are instantiated by constant processes to produce a function $OP(P)$, we have

$$OP(\sqcap S) = \sqcap\{OP(P) \mid P \in S\}$$

This equality holds in behavioural models and is closely related to the fact that processes' representations in these are derived from linear, not branching, observations. It does *not* hold up to bisimulation over the operational semantics.

It is elementary to show that any composition of distributive functions which uses each argument only once is itself distributive. What this means is that if a term is constructed in a language in which a given argument appears once, and every operator in the path that leads from it to the root of the syntax tree is distributive, then the term is itself distributive in that argument.

Our simulations $SOP(m, \psi, \chi)[\mathbf{P}, \mathbf{Q}]$ are trivially distributive in each component $P_r$ of $\mathbf{P}$ (namely each **on** argument of the CSP-like operator being modelled) by its process structure: $TO[\cdot]$, renaming, parallel and hiding are all distributive.

However, the process $Resources(I, \mathbf{Q}, m))$ is *not* distributive in the components of $\mathbf{Q}$. This is because (both in a single step and via recursion) it can put multiple copies of a given $\mathbf{Q}(i)$ in parallel[14]. It follows that CSP-like operators are not, in general, distributive in their **off** arguments. Indeed, the example operator *Farm* quoted earlier is not distributive: if $P = \bigsqcap\{a \to STOP \mid a \in A\}$ then it is clear that $Farm(P) \setminus \{start\}$ can perform any trace of $A$ actions, but $\bigsqcap\{Farm(a \to STOP) \setminus \{start\} \mid a \in A\}$ can only perform traces where all actions are identical.

In order to ensure that a CSP-like operator is distributive in an **off** argument, it is necessary that in every possible execution path that argument is turned on at most once. Namely, at most one copy is turned on on each step of $Resources(I, \mathbf{Q}, m))$, and once it has turned on it disappears from $range(\chi)$ in the state of *Reg*. This is the case for all **off** arguments of the standard CSP operators considered in Section 2.

Note that the distributivity of CSP-like operators in **on** arguments was *necessary* for our simulations to work. Though it is not necessary for the other arguments, it provides an interesting justification and explanation for Hoare's espousal of distributivity as a principle of CSP operators.

## 7 Ramifications for the present and future of FDR

There is nothing to prevent the simulations we have described being implemented directly in $\mathrm{CSP}_M$. The only changes necessary are firstly to give a channel name to the tuples used to represent rules and to choose representations of the functions $\phi$, $\psi$ and $\chi$ that allows them to be tested for equality. The obvious way of doing the latter is to represent a (partial) function as a set of pairs.

Of course if the operator(s) one was building did not use all of the features we built into our simulation, we could simplify the latter appropriately. The simpler it was, the more likely it would be to be practical.

In fact, essentially the full simulation described in Section 4 was implemented in an Oxford undergraduate student project [6]. For the reasons set out below it was not practical in a generally usable sense, but did prove a lot less inefficient than the present author suspected might be the case.

[14]The fact that $P \underset{A}{\|} P$ is not distributive in $P$ is one of the most basic facts about CSP. For example, the synchronisation of $(a \to STOP) \sqcap (b \to STOP)$ with itself can deadlock immediately; but those of $a \to STOP$ and $b \to STOP$ separately cannot.

There are two great obstacles to the use practical use of our simulation techniques. One is the size of the alphabets of event it creates in the extended $\Sigma$: just think how many partial functions $\phi$ alone there might be event before being multiplied by the other components of the tuples representing transitions. Someone attempting to use our techniques might therefore do well to avoid calculating events that are not used, and perhaps finding some representation that uses less: perhaps the fact that our result shows that a given operator *can be* represented will inspire programmers to find other, more efficient representations.

The second, which became particularly apparent in the work reported in [6], is the interaction of our simulation and recursion in the context of FDR, because of the particular way FDR works.

FDR deals with a term representing a CSP process by identifying a (typically parallel) recursion-free combination of *low-level* processes. It then compiles these low-level processes into finite state machines by calculating each of their operational semantics, together with a recipe for running them together. For details see [19], for example. This means that any simulation which involves our *Resources* process cannot work, since that is intrinsically infinite-state. It follows that an FDR-compatible *Resources* must have a (small) bound on turning on arguments: indeed, it is better to follow the incomplete approach set out at the start of Section 4.3.

A more subtle, and practically more damaging, problem comes when a simulated operator becomes part of one of the low-level components, and in particular when one of then is recursed through. Consider the recursive terms

$$\begin{array}{rcl} P & = & a \to (P \,\Box\, (b \to P)) \\ & & \Box\; c \to P \\[1ex] Q & = & (b \to \mathit{STOP}) \;|||\; (a \to Q) \\[1ex] R & = & \mathit{STOP} \;|||\; (a \to R) \end{array}$$

Here $|||$ is the CSP interleaving operator equivalent to $\underset{\emptyset}{\|}$. As we explore the first of these there are very clearly only finitely many states found because as soon as a visible event occurs the top-level $\Box$ operator is resolved: the operator itself disappears from the term. FDR finds this process easy to compile into a finite state machine. There is a difference in the second: the more $a$ events occur, the more complex the resulting term becomes. An infinite number of different terms are discovered as this process is explored and so FDR can never compile it into a finite state machine. In this example this is fundamental: there is no finite state equivalent to this process which effectively counts how many $a$'s have happened and allows exactly that many $b$s.

The third is obviously equivalent to the finite-state process $AS = a \to AS$: it simply communicates an arbitrary number of $a$s. However as FDR explores its state space it will discover parallel combinations of arbitrary numbers of $STOP$ with $S$. Therefore it will not succeed in compiling $S$ into its finite-state representation.

Now imagine that in place of the $\Box$ in the first example you have simulated it using the techniques set out in Section 4.2. Then $P \Box Q$ is replaced as the parallel combination of three processes: a modified $P$, a modified $Q$ and the regulator. FDR, in exploring this simulation will never eliminate this parallel combination as it can put the original $\Box$ behind it in exploring the original $P$. It will behave rather like the $S$ recursion above, and FDR will never succeed in compiling the recursive process: recursing through the simulation will generate an infinite number of different pieces of syntax.

[6] developed ways of resolving this problem in limited circumstances, but the problem of "closing" the calculation of recursive component processes in the context of parallel composition probably means that, when used directly in CSP$_M$ for FDR, the implementations/simulations of operators we have developed should only be used syntactically above the level of recursions.

Nevertheless the existence of the simulations implies, as pointed out in Section 6, that any CSP-like operator has a well-behaved semantics over any CSP model, namely the one represented by its simulation. It follows that any such operator can be added to CSP$_M$ and implemented as primitive in FDR and we can guarantee that this will make mathematical sense.

Consider, for example, the operator $P[T]Q$ defined with two **on** operands via the rules

$$((a,a),a)[a \in T] \qquad ((a,\cdot),a,\mathbf{1})[a \notin T] \qquad ((\cdot,a),a,\mathbf{2})[a \notin T]$$

$$((\checkmark,\cdot),\checkmark) \qquad ((\cdot,\checkmark),\checkmark)$$

This is a sort of hybrid between parallel and external choice: events in $T$ can only occur when both perform them and do not resolve the choice. When $T = \emptyset$ it is equivalent to $\Box$, and $[\Sigma]$ is equivalent to $\underset{\Sigma}{\parallel}$ for $\checkmark$-free processes. Events not in $T$ can be performed independently by either argument and immediately resolve the choice. One instance (and the motivating one) of this is when we are in world of timed systems with the passage of time represented by the event *tock*. Then, as described in Chapter 15 of [19] and [11], $P[\{tock\}]Q$ is the best representation of the untimed operator $P \Box Q$ and is necessary for a faithful representation of Timed CSP [12] into a form that can be analysed on FDR.

One interesting use of $[T]$ is to create a *time-out* operator: $P[\{tock\}]\,WAIT\ n$ (where $WAIT\ n$ performs exactly $n$ *tock*s before $\checkmark$) allows $P$ and the environment $n$ time units to perform a visible event, and if not forces termination.

The author believes that it is impossible to simulate $[T]$ in a general way in CSP such that recursing through it will ever create a program that FDR will succeed in terminating. Certainly the simulation this paper generates will fail for exactly the reason set out above.

The existence of the combinator operational semantics in itself, however, tells us that it is safe to add this operator into CSP$_M$ and implement it directly in FDR. While it is then possible to write instances of its use where compilation does not terminate, namely ones where in terms like $\mu\, p.F(p)[T]Q$, $F(p)$ can start its argument $p$ before the choice has been resolved. At the time of writing an implementation of this operator is in preparation.

We can be more ambitious than this, however, and look to redesigning FDR so that *all* operators are described to it solely via a combinator operational semantics. We could then make this interface available to users who wished to add new operators into its arsenal, safe in the knowledge that any such operator makes sense in all CSP models. This would be enormously more concise than the way in which operators need to be introduced into FDR at present (only via programming its source code) in different versions for compilation, high-level running, debugging etc.

As described in Chapter 9 of [19], combinator operational semantics are very close to the way in which FDR implements high-level combinations, namely the combinations of compiled state machines referred to above. It is also easy to infer a corresponding SOS operational semantics from a combinator semantics, which is essentially what the low-level compiler uses.

## 8 Examples

We have seen two CSP-like operators that are not in the usual CSP language, namely $[T]$ and $Farm(\cdot)$. A number of other examples can be found in [19] including

- An angelic version of the external choice operator: $P \boxdot Q$ behaves like either $P$ or $Q$ and offers the choice of their initial events. However, unlike $P \mathrel{\Box} Q$, the choice of an event common to $P$ and $Q$ does not resolve the choice, and they both persist to offer their choices on the next step (and so on). In other words if $P$ and $Q$ are both $\checkmark$-free deterministic processes then $P \boxdot Q$ is deterministic with the union of $P$'s and $Q$'s traces.

This turns out to have much in common with the parallel operator: the combinator operational semantics always, except on termination, keeps both arguments **on**. It does not discard $P$ or $Q$ if the other has performed some $a \in \Sigma$ or even some longer trace $t$: it gives the other the chance to catch up, and if it does perform catch-up events these become $\tau$s. We therefore need to define a whole family of binary operators: in addition to $P \boxdot Q$, $P_s \boxdot Q$ and $P \boxdot_s Q$ for each $s \in \Sigma^* \setminus \{\langle\rangle\}$, representing the states where respectively $P$ and $Q$ have to catch up by $s$. The combinators are:

- For $\boxdot$: $((a,.), a, \mathbf{1} \boxdot_{\langle a\rangle} \mathbf{2})$ and $((.,a), a, \mathbf{1}\ {}_{\langle a\rangle}\boxdot\ \mathbf{2})$ for all $a \in \Sigma$, plus $((\checkmark,.),\checkmark)$ and $((.,\checkmark),\checkmark)$.
- For ${}_{\langle b\rangle\hat{}s}\boxdot$: $((b,.), \tau, \mathbf{1} \boxdot_s \mathbf{2})$ and $((.,a), a, \mathbf{1}\ {}_{\langle a,b\rangle\hat{}s}\boxdot\ \mathbf{2})$ for all $a \in \Sigma$, plus $((\checkmark,.),\tau,\mathbf{2})$ and $((.,\checkmark),\checkmark)$.
- For $\boxdot_{\langle b\rangle\hat{}s}$: $((.,b), \tau, \mathbf{1} \boxdot_s \mathbf{2})$ and $((a,.), a, \mathbf{1}\ {}_{\langle a,b\rangle\hat{}s}\boxdot\ \mathbf{2})$ for all $a \in \Sigma$, plus $((.,\checkmark),\checkmark)$, and $((\checkmark,.),\tau,\mathbf{1})$.

• A mobile version of the parallel operator in which communications between processes can affect their alphabets and therefore which subsequent communications they synchronise on: see Section 20.3.

In [18] it is shown that the parallel operator of CCS [9] and the $\pi$ calculus is CSP-like, and indeed that for the syntax set out in [21] the whole $\pi$-calculus is CSP-like. As already remarked, the CCS $+$ operator is not CSP-like as it is resolved by an argument's $\tau$ rather than simply promoting it. [The usage of $+$ in the version of $\pi$-calculus in [21] is sufficiently limited that it is not an operator on terms, only guarded terms, and this permits a work-around.]

The semantics of $\pi$-calculus given in that paper depends on a much generalised renaming operator, in which the renaming applied varies arbitrarily with the trace. A somewhat simplified simulation of that is discussed there.

The author recalls Hoare saying many years ago that it would be elegant to allow processes to disappear from certain sorts of parallel composition when they terminate and in particular that $P \gg \mathit{SKIP}$ and $\mathit{SKIP} \gg P$

should both be equivalent to $P$. [Recall that the operator $P \gg Q$ takes two processes whose alphabet is $\{| \, \mathit{left}, \mathit{right} \, |\}$ and connects $P$'s outputs (*right*) to $Q$'s input (*left*), hiding them and creating a further process with the same alphabet. In its standard form, which does not have the termination property outlined above, it is written in terms of renaming, hiding and parallel.]

In other words Hoare wanted a process simply to disappear from a chain of processes when it terminates. As our final example we will show how to create a combinator operational semantics for the revised operator $\gg^{\checkmark}$: as with most such exercises this is remarkably easy. There are of course two arguments,both of which are **on**.

The "business end" of the modification is the pair of termination rules, which are identical to those of $\Box$:

$$((\checkmark, \cdot), \tau, \mathbf{2}) \qquad ((\cdot, \checkmark), \tau, \mathbf{1})$$

These are, or course, a little simpler than the corresponding termination rules one could infer for the usual formulation of $\gg$, with the $\Sigma$ rules being identical to that case:

$$((\mathit{left}.x, \cdot), \mathit{left}.x) \qquad ((\mathit{right}.x, \mathit{left}.x), \tau) \qquad ((\cdot, \mathit{right}.x), \mathit{right}.x)$$

The existence of a CSP simulation of this form of the operator therefore follows, though it is far from easy to see how to build one in elementary fashion. Note that one of $P$ and $Q$ terminating, some communications of the other that used to be synchronised and hidden now become unsynchronised and visible.

To see why this is an attractive operator, compare the two recursions

$$\begin{array}{rcl} B1 & = & \mathit{left}?x \rightarrow (B1 \gg \mathit{right}!x \rightarrow \mathit{COPY}) \\ B2 & = & \mathit{left}?x \rightarrow (B2 \gg^{\checkmark} \mathit{right}!x \rightarrow \mathit{SKIP}) \end{array}$$

These are both processes that satisfy the failures-divergences buffer specification given in [15] etc. Indeed both are equivalent to the unbounded determinininistic buffer defined

$$\begin{array}{rcl} B^{\infty}_{\langle\rangle} & = & \mathit{left}?x : T \rightarrow B^{\infty}_{\langle x \rangle} \\ B^{\infty}_{s\hat{\ }\langle y \rangle} & = & (\mathit{left}?x : T \rightarrow B^{\infty}_{\langle x \rangle\hat{\ } s \hat{\ }\langle y \rangle} \\ & & \Box \; \mathit{right}!y \rightarrow B^{\infty}_{s}) \end{array}$$

In the first there are always as many empty $\mathit{COPY}$ processes as there have been outputs so far. Therefore even if this process never holds more than one item, it will still grow into an unboundedly long chain as more and more items are inserted and removed. The second, however, is always a chain of exactly one more process than there are items presently in the buffer: therefore it keeps itself a lot tidier!

We conclude this section with a further example of an operator that is not CSP-like. The $\mathit{alt}(P, Q)$ operator allows $P$ and $Q$ to perform visible events in strict alternation, starting with $P$. It is easy to create a CSP simulation that looks correct, at least in the absence of termination:

$$((P \underset{\emptyset}{\|} Q[\![\mathit{prime}]\!]) \underset{A \cup A'}{\|} \mathit{Reg})[\![\mathit{unprime}]\!]$$

where $A$ and $A'$ are, as usual, disjoint copies of the processes' alphabet, and

$$\mathit{Reg} = ?x : A \to ?x : A' \to \mathit{Reg}$$

This does not, however, correspond to the ($\checkmark$-free case of) the SOS semantics

$$\frac{P \xrightarrow{\tau} P'}{\mathit{alt}(P, Q) \xrightarrow{\tau} \mathit{alt}(P', Q)} \qquad \frac{P \xrightarrow{\checkmark} P'}{\mathit{alt}(P, Q) \xrightarrow{\checkmark} \Omega} \qquad \frac{P \xrightarrow{a} P'}{\mathit{alt}(P, Q) \xrightarrow{a} \mathit{alt}(Q, P')}$$

because the latter completely suspends the second argument, while the simulation above allows it to perform $\tau$. This operator fails to be CSP-like for the same reason as $\mathit{suspend}_a(P)$ quoted earlier. As with that operator, and unlike $+$, this operator does make sense over CSP models.

## 9 Conclusions

We have shown that CSP is universal for expressing operators, satisfying a restricted but central conditions, over LTSs. The immediate corollary of this is that any language whose non-recursive constructs satisfy these conditions has a semantics in each of CSP's many models, and for each a theory of refinement.

It is clear that whenever such a language contains CSP, then each such model has exactly the same *full abstraction* property (if any) that it does for CSP.

We have not shown that our "CSP-like" conditions are the most general such, and indeed it is clear that they are not: consider the $\mathit{suspend}_a$ and *alt* operators alluded to earlier. The author doubts that there are closed-form conditions on operational semantics, extending our own, that can characterise the property of leading to well-defined operators over all CSP models.

Of course our results here suggest that one might try to find other languages, perhaps extensions of CSP, that would capture weaker conditions on operational semantics that are faithful to some other notion of process equivalence but not to all CSP models. The latter might be a specific CSP model such as traces or refusal testing, or some notion of bisimulation. An interesting form of bisimulation to consider, since (unlike weak bisimulation) it is finer than all CSP models, is *divergence-respecting weak bisimulation*, the coarsest relation that is both a weak bisimulation and does not identify an immediately divergent process with a stable one.

Rather than conjecture specific results here, we leave this to future research.

The major practical benefit from this work is probably the realisation that one can reconstruct CSP-based tools such as FDR so that they are programmable by combinator operational semantics, which makes them applicable to any CSP-like language. The author believes that this will make both these tools and the CSP notions of equivalence and refinement more useful.

## References

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

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

[4] 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.

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

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

[7] 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.

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

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

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

[11] J. Ouaknine, *Discrete analysis of continuous behaviour in real-time concurrent systems*, Oxford University DPhil thesis, 2001.

[12] G. M. Reed and A. W. Roscoe *A timed model for communicating sequential processes*, TCS Vol 58, pp 249-261.

[13] A.W. Roscoe, *A mathematical theory of Communicating Sequential Processes*, Oxford University DPhil thesis, 1982.

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

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

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

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

[18] A.W. Roscoe, *CSP is expressive enough for* $\pi$, Reflections on the work of C.A.R. Hoare, Springer 2010.

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

[20] A.W. Roscoe and Zhenzhong Wu, *Verifying Statemate Statecharts using CSP and FDR*, Formal Methods and Software Engineering, LNCS 4260, 2006.

[21] D. Sangiorgi and D. Walker *The* $\pi$*-calculus: A theory of mobile processes*, CUP, 2001.

[22] R.J. van Glabbeek, *On cool congruence formats for weak bisimulation*, Proceedings of ICTAC 2005, Springer LNCS 3722.