On the expressiveness of CSP

A.W. Roscoe

Oxford University Computing Laboratory
{Bill.Roscoe@comlab.ox.ac.uk}

Abstract. We show that Hoare's CSP, with the addition of the exception-throwing operator $P \Theta_A Q$ in which any occurrence of an event $a \in A$ within P hands control to Q, can simulate any operator that has a "CSP-like" operational semantics. 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 CSP+ is a natural language to compile other notations into. We explore the range of possibilities for CSP-like languages, which include the π-calculus.

1 Introduction

This paper is dedicated to Sir Tony Hoare on the occasion of his 75th birthday, 11 January 2009.

While other languages for concurrent systems are often defined in terms of their operational semantics, the CSP approach [4, 15] 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 τ.

Nevertheless CSP has a well-established operational semantics, 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 necessary background concerning CSP and its operational semantics. In particular we describe an operator $P \Theta_A Q$ (previously described in [17]) that provides an extension in expressive power over Hoare's CSP, so we term the extended language "CSP+". Following that, we develop intuition about what it means to be a "reasonable" operational semantics from [14, 15], into a rigorous definition of the concept of a *CSP-like* operational semantics.

The main result of our this paper then follows, in which we show that any operator (or class of operators) with CSP-like operational semantics, can be simulated precisely by

operators derived from CSP+. We can thus think of CSP+ as the "machine code" into which all such languages can be compiled.

In the following section we give some examples of operators that are CSP-like and some that are not and discover, for example, that while CCS is not CSP-like, the π-calculus is.

Indeed, we are able to give a CSP semantics to main version of the π-calculus used in Sangiorgi and Walker's book on that subject [21]. That version includes the CCS + operator, the most troublesome one for translation into CSP, but only in a neatly encapsulated form that allows us to deal with it. This, therefore, leads to a wide range of new semantics for the π-calculus, each with its own compositional theory of refinement.

There is an appendix that summarises the CSP notation used in this 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]. Not only did Tony Hoare invent the notation that proves so powerful, he also provoked this work with the assertion in the paper title [3] – see Section 6.1 of the present paper. I believe that this paper shows yet again just how well chosen Tony's original CSP language was.

I had useful conversations with several people including Samson Abramsky about the π-calculus.

Michael Goldsmith found an error in an earlier definition of the operator OF.

The work reported in this project was sponsored by EPSRC project ????.

This paper was completed while I was visiting the UNU International Institute of Software Technology in Macao, China.

2 Background

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. Whenever we refer to bisimulation in this paper we mean strong bisimulation.

Readers interested the abstract, behavioural models of CSP should consult [15–17]. 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*).

In much of this paper we will need to make major extensions to the alphabet of event names that processes are defined over. We will assume that the basic language of processes about which we are reasoning has visible event names drawn from a set Σ_0, but we will find ourselves extending this to much larger $\Sigma \supseteq \Sigma_0$. (However, unless Σ_0 is finite, Σ will always have the same cardinality.) We will assume that all the additional sets added into the alphabet are disjoint from Σ_0.

2.1 The CSP+ language and its operational semantics

The language we use in this paper was termed CSP+ in [17], because there is an extra operator that strictly extends its expressive power beyond the language described by Hoare [4]. As in [17], we remove the two CSP constructs (*SKIP* and ;) that give a special role to the event ✓. This is mainly to avoid the special cases and more elaborate programming that would be necessary if we maintained them, arising out of the nature of ✓ as a *signal* event. It ought to be possible to rework the definitions and results of this paper to include them, if desired[1].

We list the operators of this language below, giving a natural-deduction style operational semantics as an LTS – the form we will always assume for operational semantics in this paper. All of these operational semantics, apart from Θ_A, are taken or derived from [15].

In the following rules, x stands for any action (visible or τ), whereas a stands for only visible actions.

First, some constant processes:

- *STOP* which does nothing – a representation of deadlock. It has no semantic clauses since it has no actions.
- **div** which performs (only) an infinite sequence of internal τs:

$$\frac{}{\mathbf{div} \xrightarrow{\tau} \mathbf{div}}$$

  This is a representation of divergence or live-lock.
- $CHAOS(A)$ which can do anything except diverge within the set of events A. The following definition works in all the standard models of CSP:

$$\frac{}{CHAOS(A) \xrightarrow{\tau} ?x : B \to CHAOS(A)}(B \subseteq A)$$

  In other words, $CHAOS(A)$ can decide to offer the environment any subset of A including A and the empty set $\emptyset$ (the latter meaning deadlock).
- $RUN(A)$ which always offers the actions A.

$$\frac{}{RUN(A) \xrightarrow{a} RUN(A)}(a \in A)$$

CSP has a variety of *prefixing* operators that allow either on) or a choice of initial communications, followed by subsequent processes. [15] collects all of these into a single operational rule:

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

where $comms(e)$ is the set of communications allowed by the prefix and $subs(a, e, P)$ represents the process of selecting the correct subsequent behaviour. (P may here be a term with some non-process free variables that are substituted to turn it into a fully instantiated

[1] In that case we would probably have to accept a form of simulation between the operational semantics of the CSP-like operator and its CSP translation that was slightly weaker than strong bisimulation. This is because it will be necessary, as seen in some of the constructions of [16], to use $P;\ (e \to SKIP)$ in place of P in the translations. Here, e is an extra event in the extended Σ that allows the effects of P's termination to be transmitted to other processes. In such translations e will always be hidden, meaning that the net effect is putting an additional τ before the ✓.

process, or we can think of it as a function from the potential "inputs" of e to process terms.)

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

One of the most important observations about prefixing is that the initial actions of $e \to P$ do not depend on P, only on e. Whether P is a fully instantiated process or not, it is initially not running: it is *off*. It is not set running until after the initial action that is selected – in this case, by the environment.

There are only two other operators that follow this model. One of these is nondeterministic choice, which is modelled by a choice of τ actions, one to each process we are choosing between:

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

This easily translates to the generalised notion of choice over a non-empty set of processes:

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

It is important to remember the restriction that S is non-empty since even though the above rule makes sense, in itself, when $S = \emptyset$, the value of $\sqcap \emptyset$ it predicts does not. There is a fundamental objection to $\sqcap \emptyset$: in any nondeterministic choice the implementation is obliged to select one of the processes between which the choice is made – when there are no such processes this is *impossible*

The only other case where the initial actions are determined completely by the operator itself is in one of two alternative treatments that CSP gives to recursion. If we want to give an appropriate meaning to all recursive terms, even ill-defined ones such as $\mu\, p.p$ and $\mu\, p.(p \,\Box\, p)$, then we can use the rule[2]

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

in which the action of unfolding a recursion generates a τ, and therefore both of the above terms simply diverge. This τ causes no harm as far as any behavioural model is concerned, because in such models the value of any process is unchanged by the addition of an initial τ that leads unconditionally to the original starting state. In fact, for well-constructed recursions, the τ is not really needed, and indeed the FDR tool omits it because of the increased state spaces it creates – needlessly provided only well-constructed recursions are used. In subsequent sections we will define a number of processes recursively to play roles in simulations we will create: *for the purpose of establishing the various bisimilarity results we establish, it is necessary to assume that the FDR approach to the semantics of recursion is used: a recursive term unfolds without the creation of a τ.*

The inference rule for this non-τ recursion model is

$$\frac{F(\mu\, p.F(p)) \xrightarrow{x} Q}{\mu\, p.F(p) \xrightarrow{x} Q} \qquad \textit{equivalently} \qquad \frac{}{\mu\, p.F(p) \Rightarrow F(\mu\, p.P)}$$

where in the right-hand rule $P \Rightarrow Q$ means simple re-writing without generating an action that appears in the LTS.

[2] In all three versions of the rule for recursive unwinding, there is the assumption that bound identifier names other than p have been changed to avoid binding any variable free in the substituted terms.

We emphasise, however, that in CSP this rule may only be applied in cases where F always adds at least one action before starting its argument.

Imagine that the operators have some of their arguments 'switched on' and some 'switched off'. The former are the ones whose actions are immediately relevant, the latter the ones which are not needed to deduce the first actions of the combination. (All the arguments of the operators seen above are initially switched off, except for the second treatment of recursion in which we can re-phrase the validity condition as being that $F(P)$ has P as an off argument.) This idea comes across most clearly in the standard CSP operator $P;\ Q$ (which we are not considering here) and the extra operator $P \,\Theta_A\, Q$ of CSP+. In each of these the first argument P is switched on, but the second Q is not as its actions do not become enabled until after the first has terminated.

Both the arguments of the external choice operator ($P \,\Box\, Q$), which is resolved by the first of P and Q to perform a *visible* action, are necessarily switched on. Once an argument is switched on, it must be allowed to perform any τ action it is capable of, since the argument's environment (in this case the operator) is, by assumption, incapable of stopping them. This is reflected in the rules

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

which simply allow the τs to happen without otherwise affecting the process state. We can expect every "turned on" argument of an operator to give a rule like this. These rules simply *promote* the τ action of the arguments to τ actions of the whole process.

Operators can also allow their arguments to perform visible actions (i.e. members of Σ). Unlike the case with τ, we assume that the operator can prevent such actions and can change its own state as a result of one of these actions. The rules of this form for $\Box$ are:

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

An action performed by one or more of the arguments will always translate to an action of the complete construct. Here, the external action is the same as the one performed by P or Q, but this is not always so.

The rules for hiding and renaming have much in common, since both simply allow all the actions of the underlying process but change some of the names of the events. Any event not being hidden retains its own name under $\setminus B$, while those in B become τs:

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

Renaming has no effect on τ actions and applies the relation R to all others:

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

While CSP has several parallel operators, they can all be expressed in terms of the operator $\underset{X}{\|}$, which takes two processes and enforces synchronisation on the set $X \subseteq \Sigma$. Because of this we will only give operational semantics for $\underset{X}{\|}$ – all the others being deducible.

Since both the arguments are necessarily switched on, we need rules to promote τ actions:

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

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

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

and one to show that $a \in X$ requires both participants to synchronise

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

In addition, CSP uses conditional constructs which can be reduced to the infix form $P \lhd b \rhd Q$ which is equivalent to P or Q depending on whether the boolean expression takes the value *tru* or *false* respectively.

$$\overline{P \lhd \mathit{true} \rhd Q \Rightarrow P} \qquad \overline{P \lhd \mathit{false} \rhd Q \Rightarrow Q}$$

(The double arrow $P \Rightarrow Q$ means that P transforms itself to Q without generating any action that is recorded.)

We can view the above operators as being the core of Hoare's original vision of CSP[3]. The following pair are well-established extensions, with the second having been introduced by Hoare in his book [4], and the first being expressible in terms of the core language.

For analysing the theory of CSP it is enormously useful to have the direct representation of an asymmetric choice operator $P \rhd Q$ that is equivalent to $(P \,\Box\, a \rightarrow Q) \setminus \{a\}$ for a an event not appearing in P or Q. In other words it is allowed to offer initial events of P, but if none is taken up quickly it will perform a τ and behave like Q. The first rule promotes P's τs:

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

Any visible action from P decides the choice in its favour

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

while at any moment the combination can "time out" and become Q.

$$\overline{P \rhd Q \xrightarrow{\tau} Q}$$

The *interrupt* operator $P \triangle Q$ allows P to proceed, but at any time (not just initially), if an event of Q occurs then P is discarded and Q proceeds. Initially both arguments are turned on:

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

[3] Hoare did not use the generalised form of parallel given here, though that can be derived from operators he did introduce, namely alphabetised parallel and renaming.

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

It has long been known (see [15, 16]) that $P \triangle Q$ has a special place in CSP because of the way it can make the unstable traces of P relevant to the stable behaviours of the result – thanks to P being interrupted by Q and discarded. The power of this operator is further illustrated by the following result, which is both of interest as an example of how to translate constructs into CSP, and shows that $\Box$ could actually be removed from the language without changing its expressive power.

Lemma 1. *The external choice operator $P \Box Q$ can be simulated, up to operational semantic bisimulation, by interrupt, renaming and parallel.*

PROOF We use a fairly common trick and assume that our alphabet Σ_0 has been extended by the disjoint set $\Sigma_1 = \{a' \mid a \in \Sigma_0\}$ where the priming operation a' is injective. Let *Prime* and *Unprime* respectively be the renamings that map every $a \in \Sigma_0$ to a', and every $a' \in \Sigma_1$ to a, in each case acting as the identity on the rest of their domains. We can now define

$$P \Box^\dagger Q = ((P \triangle RUN(\Sigma_1)) \underset{\Sigma_0 \cup \Sigma_1}{\parallel} (Q[\![Prime]\!] \triangle RUN(\Sigma_0)))[\![Unprime]\!]$$

In $P \Box^\dagger Q$ both P and Q may perform initial τs as long as they please, but as soon as one of them performs a visible action it discards the other through forcing one or other RUN to act. This is clearly bisimilar to $P \Box Q$, with, for example, any state P' that the latter can reach after P performs a visible action corresponding to

$$(P' \triangle RUN(\Sigma_1)) \underset{\Sigma_0 \cup \Sigma_1}{\parallel} (RUN(\Sigma_0)))[\![Unprime]\!]$$

(Note that P' will, in fact, never be interrupted from this state and that $RUN(\Sigma_0)$ agrees to all its actions.) ∎

In [17] it was observed that none of the operators we have discussed so far has the capability of allowing one operand to turn itself off – discard itself – through performing a visible action. This had the consequence – crucial in [17] – of meaning that any such operator must transmit the divergence of an operand after performing any given action to the outside world whenever that action is performed. In that paper, therefore, an *exception throwing* operator $P \Theta_a Q$ was introduced in which the first occurrence of the action a in P has the effect of discarding P and starting up Q. In [17], the overall alphabet was assumed to be finite, meaning that Θ_a (triggered by a single event) could easily express Θ_A (triggered by any member of the set A). Since we wish to allow infinite alphabets, we need to generalise the use of this operator so that $P \Theta_A Q$ has any member of the *set* of events A throw control to Q.[4]

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

It was shown in [17] that $\triangle$ can be simulated by a construction involving Θ_a and core operators, in much the same way that we showed above that $\Box$ can be derived from $\triangle$ etc.

[4] A further generalisation would be to allow, as in $?x : A \to Q(x)$, the event to control *which* process to throw control to. It is reasonably straightforward to define this further generalisation in terms of Θ_A, parallel composition and renaming.

It seems to the author that Θ_A is an operator entirely within the spirit of [4], where indeed much space was devoted to exception handling, but for some reason only exceptions generated from outside a process rather than from within.

3 CSP-like operators

Normally, in determining whether an operational semantics for a CSP operator is valid, one has in mind a semantics in terms of one or more behavioural models. There are certainly a number of properties that the valid CSP operational semantic definition have in common so as to achieve consistency In this section we attempt to capture the essence of what makes operational semantics "CSP-like", by collecting the intuitions developed from the definitions and discussion in the previous section.

The main message one can derive from the previous section is that a CSP operator controls its operands in the same way that an external interacting environment might: turning them on, interacting with them one visible actions, and eventually discarding them perhaps. Furthermore, when an operator is turned on, it can perform τ actions without the operator being able either to control or see them.

The arguments of any operator are thus **on** or **off**: this is how we will write these types henceforth.

The operator can perform external (e.g. prefixing) or internal (e.g. $\sqcap$) actions without their operands being **on**. When an operand is **on**, the operator might transmit its visible actions directly to the outside (e.g. $\Box$) or transform them into other visible (e.g. renaming) or invisible (e.g. hiding) actions. Such a result action may be involve a single argument process or more than one of them synchronising (e.g. $\|$).

Only **on** operands can perform actions in any given state, but an action can turn an argument on or, as discussed in the previous section, discard an operand.

The structure of the resulting process of performing an action will always be the same if it a promoted τ action, but may change if it is not. For example, $P \Box Q$ is transformed into something of the form $P' \Box Q'$ by τs of P and Q, but to the simple form P' or Q' by a visible action.

The intuition above is essentially copied from the author's doctoral thesis [14] (page 233 *et seq*). What we now do is to formalise it (26 years later). In the following definition we try to capture the properties of an arbitrary combination $C[P_1, P_2, \ldots, P_n]$ of processes under this intuition. Specifically we want a definition that covers all combinations of CSP operators. A little thought reveals that in any such combination there are only finitely many processes turned **on**: this can be proved by induction, since no single action can turn on an infinite set of arguments. This is just as well, since an infinite number of arguments all performing just one τ each would lead to divergence of the whole, contradicting our desired principle that $\tau.P$ should be indistinguishable, in abstract models, from P. On the other hand both infinite nondeterministic choice and, semi-implicitly, prefixing in the context of an infinite alphabet, envisage infinite collections of **off** operands. In these cases at most one will ever be turned on. We will therefore consider any operator to be a formal function of some finite ordered list, with a given length n, of **on** operands and a set Π indexed by some I of **off** ones.

The type of such an operator will therefore be a pair (n, I) of these two "arities". In general it is natural to think of a CSP operator as having other arguments: events, sets, renamings and other parameter values. However in this initial treatment of this subject we will think of these as given constants, so each combination of valid parameters of this

sort will give a different "operator" in the formal sense. Thus, for example, $\underset{X}{\|}$ will be many different operators as X varies, all of arity $(2, \emptyset)$.

The operational semantics of any operator is a set of rules, under which combinations of zero or more actions of the **on** arguments (no more than one per process) yield both an action of the whole, and a successor state which is itself a construction based on the arguments of the original operator.

We will permit only *positive* rules: namely no action or result state ever depends on the absence of any action. Furthermore the use of the arguments in the result state arising from a given rule will be strictly governed as follows:

- Any **off** argument that remains an argument (whether **on** or **off**) will be in its initial state.
- Any **on** argument that has contributed the action $P \xrightarrow{x} P'$ to the rule will be in state P' if it is present.
- Any **on** argument P that has not contributed any action will be in state P if it is present.
- The rule does not depend on any argument performing more than one action, either as alternatives or in sequence.

The rules belonging to each operator will each contain a partial function from the **on** argument indices to actions, and the condition for that rule firing is that the corresponding processes can perform the given action. When the domain of this function is empty it corresponds to the operator itself performing the action corresponding to the rule independently of its arguments. There may be any number of rules with a given function: the result states being different. Each rule also has a representation of the action that results from that rule "firing": this action may be a visible action on Σ or it may be τ.

The only restriction on what rules of this sort an operator may have is the following:

- There is one rule for each **on** argument $i \in \{1, \ldots, n\}$ representing the promotion of a τ without otherwise influencing the state:

$$\frac{P_i \xrightarrow{\tau} P_i'}{OP(P_1, \ldots, P_i, \ldots, P_n, \Pi) \xrightarrow{\tau} OP(P_1, \ldots, P_i', \ldots, P_n, \Pi)}$$

  No other rule has a τ as a contributing action (i.e. in the range of the partial function described above). This restriction corresponds to the idea that operators are not aware of the τ actions of their operands, and cannot take any positive action on account of these.

We regard the statement above as one of two restrictions that are crucial in distinguishing "CSP-like" operators and languages. Clearly it excludes the CCS + operator, which allows a τ action to resolve choice and therefore requires the operator itself to be aware that a τ has taken place. The other restriction will follow shortly.

For the other rules, we need to decide on the format we will allow for the "result states" of an action. There seem to be two reasonable alternatives for the overall appearance of this format:

1. The result state is a single operator (possibly the identity operator) applied to processes derived in an appropriate way from the arguments of OP.
2. It might be a well-founded composition of such operators: in other words a syntax tree with no infinite paths, in which some of the arguments to operators are the drawn from the arguments to OP and some are other such terms.

Let us first consider the restrictions we should place on version 1. Firstly, an **on** argument of OP can only appear as an **on** argument of each result operator OP': we do not allow **on** arguments to be suspended.

Secondly, each **on** argument can only appear at most once amongst the arguments of OP'. This represents the second main principle underlying *CSP-like* semantics: we do not allow a process which is up and running to be cloned and then possibly compared against itself. It is well known that this restriction is necessary if we are to model processes by descriptions of individual linear runs – into which category fall all the types of behaviour used in the CSP modelling approach.

This CSP approach leads naturally to the principle that the simple prefix operator $a \rightarrow P$ distributes over nondeterministic choice $\sqcap$. For example, the linear observations of $P_1 = a \rightarrow ((b \rightarrow STOP) \sqcap (c \rightarrow STOP))$ and $P_2 = (a \rightarrow (b \rightarrow STOP)) \sqcap (a \rightarrow (c \rightarrow STOP))$ are identical when τ cannot be seen: the observer sees a offered, performed if accepted by the observer, and then either a behaviour of $b \rightarrow STOP$ or of $c \rightarrow STOP$. On the other hand, if we had an operator that could clone the behaviour just after the initial a and then synchronise the two copies, then for P_2 we can guarantee that the two copies will both perform b or both perform c. This would be because the choice between b and c must already have been made by the time that a occurs. On the other hand, P_1 might delay the choice between b and c until after the cloning, meaning that the two copies might choose differently. Therefore the operator applied to P_1 could deadlock immediately after the cloning, but applied to P_2 it cannot.

The arity of OP' will be some (m, J) possibly different to (n, I). The arguments of OP' indexed by $\{1, \ldots, m\}$ and J can come from three sources:

(i) The **on** arguments of OP, subject to the above restriction, with each process either being some P_r or the result P'_r of P_r performing the action leading to this successor state. Only the **on** arguments of OP' can take this form.
(ii) The **off** arguments of OP, with the corresponding process being exactly the argument taken from OP.
(iii) A constant process defined independently of the arguments of OP.

There is, in fact, no need to make explicit provision for (iii) in what follows, since we can include such constant processes as extra **off** arguments and extend the corresponding indexing set I accordingly.

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

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

can actually copy the argument P arbitrarily often. (This one just interleaves a "farm" of as many Ps as *start* events have occurred – assuming that *start* is not an event of P.) There is no fundamental 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 $Farm_n(Q_1, \ldots, Q_n, P)$ which runs Q_1 to Q_n in parallel (all as **on**) and also offers *start* independently of its arguments before moving to a $Farm_{n+1}$ state. Thus P is an **off** argument: $Farm = Farm_0$, where

$$\frac{}{Farm_n(Q_1, \ldots, Q_n, P) \xrightarrow{start} Farm_{n+1}(Q_1, \ldots, Q_n, P, P)}$$

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

We have opted not to allow processes to be suspended once they are **on**: as long as they remain part of the state they remain active, in the sense that they can perform τ actions at least. Notice that if we *did* allow this, we could not simply place the suspended arguments amongst the **off** arguments, as to copy such a process and then restart it would give exactly the same problems as copying an **on** one.

This suggests an element of caution would have to be exercised if we were to adopt option (2) above, where the result state is a tree of operators. It indicates that we need to restrict the use of the **on** arguments of the original OP to use in places where there is no **off** argument in the path from the use to the root of the tree. In other words, you should never place an **on** argument of OP in any **off** place in the tree, even as a part of a larger object. None of the **on** arguments can appear at more than one place in the resulting tree.

We can think of any such syntax tree of operators as a single operator in its own right, and the restrictions we have placed above mean that the whole tree only has finitely many **on** arguments (namely ones that are **on** in every operator on the path from the route). Furthermore the transitions of the tree can be calculated by recursion over the **on** arguments of the various sub-terms, and necessarily lead to another tree satisfying our conditions. The empty tree of operators can be represented by the identity operator (with a single **on** argument and which simply promotes all actions of this argument). For this reason we observe that the apparently more general second approach gives no more expressive generality at all, though it may allow only a finite class of operators to be defined for expressing the same semantics that requires an infinite set using the first approach. (An example of this is the *Farm* operator defined above.)

We therefore adopt the first version as our standard way of presenting the operational semantics of an operator.

The descriptions in this section to date motivate the following formal definition of the operational semantics of a class of operators $\{OP_\alpha \mid \alpha \in A\}$. For each $\alpha \in A$ there are arities $n(\alpha) \in \mathbb{N}$ and $I(\alpha)$ of **on** and **off** arguments respectively. We will assume for convenience that the indexing set $I(\alpha)$ is disjoint from the natural numbers $\mathbb{N}$. The first step behaviour of $OP_\alpha(\mathbf{P}, \mathbf{Q})$ ($\mathbf{P}$ and $\mathbf{Q}$ being respectively vectors of processes representing the **on** and **off** arguments) are precisely those deducible by the following two classes of rule:

- There are the $n(\alpha)$ rules that promote τs. Since these rules are always present there is no need to record them in the formal representation of an operator.
- There is an arbitrary set of rules that do not have τs as premises. Each such rule is represented as a tuple $(\phi, x, \beta, f, \psi, \chi)$ where
  - ϕ is a partial function from $\{1, \ldots, n(\alpha)\}$ to Σ_0 (the alphabet of the underlying processes). 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 $\Sigma_0 \cup \{\tau\}$ that $OP_\alpha(\mathbf{P}, \mathbf{Q})$ performs as a consequence of the condition expressed in ϕ being satisfied.
  - β is the index of the operator 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.

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

To illustrate this way of representing operators, we will show how some of the CSP operators fit into this framework. None of them, in fact, need to be defined together with any other operator apart from the identity **id** whose arity is $(1,\emptyset)$ and which has the rules $\{(\{(1,a)\},a,\mathbf{id},\emptyset,\{(1,1)\},\emptyset) \mid a \in \Sigma\}$. Here we have represented the (partial) functions as sets of pairs.

Given this we can now define:

- $a \to \cdot$ has arity $(0,\{-1\})$ and the single rule $(\emptyset,a,\mathbf{id},\{(1,-1)\},\{(1,1)\},\emptyset)$. We have used a negative number to index the **off** argument as this is a convenient way of making sure they are disjoint from the **on** indices. **id** is (the index of) the identity operator. Thus here $k=1$ and the resulting operator has no **off** arguments.
- $\Box$ has arity $(2,\emptyset)$ and the rules $(\{(1,a)\},a,\mathbf{id},\emptyset,\{(1,1)\},\emptyset)$ and $(\{(2,a)\},a,\mathbf{id},\emptyset,\{(1,2)\},\emptyset)$ for each $a \in \Sigma$.
- $\setminus X$ has arity $(1,\emptyset)$ and the rules $(\{(1,a)\},\tau,\setminus X,\emptyset,\{(1,1)\},\emptyset)$ for all $a \in X$ and $(\{(1,a)\},a,\setminus X,\emptyset,\{(1,1)\},\emptyset)$ for all $a \in \Sigma - X$.
- $\underset{X}{\parallel}$ has arity $(2,\emptyset)$ and rules $(\{(1,a),(2,a)\},a,\underset{X}{\parallel},\emptyset,\{(1,1),(2,2)\},\emptyset)$ for all $a \in X$ and both $(\{(1,a)\},a,\underset{X}{\parallel},\emptyset,\{(1,1),(2,2)\},\emptyset)$ and $(\{(2,a)\},a,\underset{X}{\parallel},\emptyset,\{(1,1),(2,2)\},\emptyset)$ for all $a \notin X$.
- Θ_A has arity $(1,\{-1\})$ and the rule $(\{(1,a)\},a,\mathbf{id},\{(1,-1)\},\{(1,1)\},\emptyset)$ for each $a \in A$ as well as $(\{(1,b)\},b,\Theta_A,\emptyset,\{(1,1)\},\{((-1,-1)\})$ for each $b \in \Sigma - A$.

Indeed, naturally, *all* the operators of CSP+ are CSP-like!

4 CSP simulation

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. We have already seen it in action in the proof of Lemma 1. In this section we will use this and other tricks to prove the following result.

Theorem 1. *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_λ 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 bisimilarity of transition systems. In other words, any operator with a CSP-like operational semantics can can be simulated in CSP+.*

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_λ that is applied to them (with all OP_λ having arity $(n, \emptyset)$, and
- a permutation π that relates the indices of the fixed argument processes to their positions as arguments of OP_λ. Because it matches our later needs better, we will record π as a function that maps each argument of OP_λ 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

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

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

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

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

as a consequence of the operational semantics of the CSP operators used to build SOP. This means that the compulsory τ-promotion rules are accurately simulated by construction. This, of course, helps to indicate *why* they are compulsory.

The assumptions we are making here imply 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 μ and π' and states P'_r of P_r. What we might therefore expect is that every state of $SOP(\lambda, \pi)[P_1, \ldots, P_n]$ will be of the form $SOP(\mu, \pi')[P'_1, \ldots, P'_n]$.

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

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 τ. 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 (ϕ,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 $\Sigma - \Sigma_0$ that represents when the combination synchronises to produce a τ. The renaming BR_i maps the event $a \in \Sigma_0$ to all such pairs (ϕ,x) such that $i \in dom(\phi)$ and $\phi(i) = a$. The alphabet A_i is then the range of CR_i. 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 ϕ 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 ϕ with an appropriate domain, including them all synchronising when ϕ 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 (ϕ,x) actions in the combination S give us a way of achieving every possible transition rule that we allowed other than the τ-promotion ones that are already accounted for. Similarly it should be clear that the final renaming CR will just map each (ϕ,x) to x, and that H (the set hidden in the definition of SOP) is $\{tau\}$.

Just as the form of our simulation *forces* τ 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_λ 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 (ϕ,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 π maps the indices of the arguments of OP_λ 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_λ 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 ϕ, this must correspond to OP_λ 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_λ must take the form $(\phi, x, \mu, \emptyset, \xi, \emptyset)$[5] where ϕ are the firing conditions, x is the resulting action, μ is the index of the resulting operator, and ξ is the permutation mapping the arguments of OP_μ to those of OP_λ. 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_μ to the participants $P_1, \ldots, P_n$ of S. We therefore define[6]

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

where $TR(\lambda)$ are the firing rules of OP_λ, 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 ϕ 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_λ does, independent of what the arguments P_i or cannot do.

Notice also that, for every single firing rule of OP_λ, 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 2. *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 μ, π 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: 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.

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

[6] 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 τ action. This is why we have adopted the semantics of recursion that does not create this type of τ.

We can therefore assume that every OP_λ in a class has n or less **on** arguments (since all those reachable from the consideration of a particular 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 τs 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 τ. 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_λ with arity n, satisfying the assumptions of this middle step. Then we can compute $\mathit{Discard}(\phi)$ 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 (ϕ, x, B) where B is the set of arguments turned off by the corresponding rule ρ: $B = \mathit{Discard}(\rho)$. The renamings BR_r need to be extended to accommodate this:

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

The first line here covers the ways P_r can proceed normally without being turned off. 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 (ϕ, x, B) to x. We note that triples of the form (ϕ, x, B) where no events participate in the action can still have $B \neq \emptyset$ and discard processes.

In $\mathit{Reg}(\mu, \psi)$, ψ 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_μ. All the original processes not in the range of ψ will have been discarded by previous actions when this state is reached and therefore become $STOP$ in the simulation.

We can therefore define

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

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 $SOP(n, \lambda, \mathbf{id})[\mathbf{P}]$, where for general $n \geq n(\mu)$, μ, $n(\mu)$-tuples of processes $\mathbf{V}$ and ψ 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 3. *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 λ. Suppose that $\lambda \in \Lambda$ and that $\psi : \{1,\ldots,n(\lambda)\} \rightarrow \{1,\ldots,n\}$ 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 τ 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 τ 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_μ.

4.3 Turning processes on

Our full definition of a CSP-like operational semantics allows processes to be held in reserve and turned on later, and for constant processes to be introduced. These processes, unlike **on** ones, may be copied, but not once they have been turned 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 (implicit copying) 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 $on \rightarrow TO(Q)$. 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 τ 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 (ϕ, 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. This is just as well, since infinite parallel composition has no definition in CSP that makes sense under our normal modelling assumptions. The processes P and $(a \rightarrow P) \setminus \{a\}$ are always equivalent in CSP when a does not occur in P: an initial τ does not change the semantics of a process. But if we placed an infinite number of processes, all with an initial τ, in parallel, then the promotion of these τs would lead to divergence, something with a very definite semantic effect.

Dynamic parallel compositions can easily be created in CSP by recursing through parallel operators. This has been done since the earliest days of CSP [2], frequently using the chaining $\gg$ 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 [14].

Thus in our case we should expect to start with a finite parallel composition including a single process containing all of the **off** arguments as well as the constant processes that operator definitions use, and make the action, that would turn on one of these, cause that process to spawn off a parallel copy of the correct argument[7] while still being available 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 combination. Fortunately, however, our previous choice of finite partial functions ϕ still works. Each Σ_0 will now be renamed to an infinite set of actions even when Σ_0 itself is finite, since the ϕs may now have any finite subset of $\mathbb{N}$ as their domains.

[7] In fact it will have to be able to spawn off any finite combination of these arguments since we allow a single action to fill all of the **on** arguments of the successor operator from this source.

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:

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

where the V_r are states relating to $TO(P_r)$ where P_r is either some initial argument or a process $\mathbf{Q}(i)$ that has been turned on, $A_r^+ = \bigcup\{A_i \mid i \leq r\}$, $A_r^- = \bigcup\{A_i \mid i > r\}$. $\mathit{Resources}(I, \mathbf{Q}, r)$ 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 (ϕ, x, B, m, f) where ϕ is a finite partial function from $\{1, \ldots, m\}$ to Σ_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).

We only use events where B and $\mathit{dom}(\phi)$ are disjoint. The role of f is to assign indexed processes to whichever new parallel slots this operator creates.

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

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

where $A^{\#}_{k,l} = \bigcup_{r=k+1}^{l} A_r$.

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. This combination, naturally, has the same alphabet as the original process $\mathit{Resources}(I, \mathbf{Q}, m)$. Provided at least one new process is spawned off, the remaining $\mathit{Resources}(I, \Pi, m+ \mid f \mid)$ has a strictly smaller alphabet.

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 $\mathbf{Q}(i)$.

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

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

with the renamings BR_r being extended to

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

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 $Resources(I, \mathbf{Q}, n)$ have the same effect as though they were combined directly with the already turned-on components.

We need to extend the controlling process *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 λ, an integer m saying how many processes have already been started up, an injective function ψ that maps $\{1, \ldots, n(\lambda)\}$ to $\{1, \ldots, m\}$ which establishes which of the m processes each of the **on** arguments of OP_λ is, and a function $\chi : I(\lambda) \to I(\lambda_0)$ which maps each index of an **off** argument of OP_λ to an index of one of the original **off** arguments: **off** arguments, by their nature, have not made any progress since the start. Recall that one of our assumptions about CSP-like operators is that all **off** arguments of a successor operator are **off** arguments of the original. λ_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.

$$\begin{aligned} Reg(\lambda, m, \psi, \chi) = \Box\{(\phi \circ \psi^{-1}, x, \psi(Discard(\rho)), \chi \circ f, m) \to \\ Reg(\mu, m+ \mid f \mid, (\psi \cup id_{\{m+1,\ldots,m+\mid f \mid\}}) \circ \psi', \chi \circ \chi') \mid \\ \rho = (\phi, x, \mu, \psi', \chi', f) \in TR(\lambda)\} \end{aligned}$$

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

- If a component has been freshly created by this transition (i.e. the result of applying ψ' to it gives a result greater than m) then it is mapped directly to the result of ψ'. Here id_A is just the identity function on the set A.
- If a component was already in existence before the rule fires (i.e. ψ' maps it to an index no greater than m) then we need to compose ψ' with the function ψ 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 ψ from $\{1, \ldots, n(\mu)\}$ to $\{1, \ldots, m\}$ and a function χ from $I(\mu)$ to I_0, we can define

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

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 (ϕ, 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 4. *Suppose we have a family of operators $\{OP_\lambda \mid \lambda \in \Lambda\}$, and that $\lambda \in \Lambda$, $m \geq n(\lambda)$, that $\psi : \{1, \ldots, n(\lambda)\} \to \{1, \ldots, m\}$ 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:*

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

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 **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 ψ and χ both being the identity function gives us Theorem 1: $C_\lambda = SOP(n(\lambda), id_{\{1...,n(\lambda)\}}, id_{I(\lambda)})$.

5 Ramifications

There is nothing to prevent the simulations we have described being implemented directly in CSP_M. The only changes we would have to make are firstly to give a channel name to the tuples used to represent rules and to choose representations of the functions ϕ, ψ and χ 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. It seems likely to the author that this approach might be practical in situations where we want to add a small number of relatively simple new operators to CSP (or a subset). One could build an "interpreter" that could take a operators described as sets of operational rules (or some representation thereof) and convert them into functions that would replicate the operational semantics of these operators.

The combination of FDR and CSP have previously demonstrated that they can be surprisingly effective and efficient in running intricate and large-scale simulations, for example [20]. The author suspects that the most significant obstacle will be the size of the sets of events involved. 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.

Our main motivation for developing Theorem 1 was, however, a theoretical one. We have shown that every operator with a CSP-like operational semantics can be implemented using the CSP+ language and nothing else. It follows that every such operator makes sense over any semantic model that is a congruence for CSP+. Thus the following theorem is actually an immediate corollary to Theorem 1.

Theorem 2. *Suppose we are given an alphabet Σ_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 τ 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 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 the 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*.

In future it may well be preferable to implement CSP-like languages by representing their operational semantics in a way that FDR can understand directly without interpreting them in CSP. In that way they would be handled in the same way that FDR currently handles CSP. The results of the present paper therefore provide an important part of the underpinning for the current effort to enable users of FDR to get direct access to the internal

We will define a language that has a denotational semantics for every CSP model $\mathcal{M}$ as *CSP-like* (as an obvious analogue of CSP-like operators). Thus the consequence of Theorem 2 is that the language $\mathcal{L}$ is CSP-like. As we will see in Section 6.3, there are CSP-like languages that do not fit exactly into the form assumed in this theorem.

5.1 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 abstract models: it does *not* hold up to bisimulation over the operational semantics.

It is elementary to show that any composition of distributive functions 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[8]. 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 $P = \sqcap\{a \rightarrow STOP \mid a \in A\}$ then it is clear that $Farm(P) \setminus \{start\}$ can perform any trace of A actions, but $\sqcap\{Farm(a \rightarrow STOP) \setminus \{start\} \mid a \in A\}$ 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 3.

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

6 Some examples

In this section we illustrate the power of Theorem 1 by illustrating both how an important new idea can be brought into CSP and other process calculi can be mapped into CSP models. There is almost certainly much more that can be said about the results of all three of the following sections, but here we concentrate on how they provide interesting examples for this paper.

6.1 CCS

If we look at the standard operational semantics of CCS [10, 11], the operator that stands out as *not* being CSP-like is +, since this can be resolved by a τ action performed by either of its operands. The operational semantics of $\alpha.P$ for $\alpha \neq \tau$ are identical to those of CSP prefix. The semantics of recursion in CCS is essentially[9] identical to the non-τ version in CSP, and the CCS relabelling operation is a case of CSP renaming. Let us consider the rest of the language: parallel $|$, and restriction $\setminus \alpha$.

The structure of Σ_0 (as we again call the set of visible action names used in creating processes) with an operator $\overline{\alpha}$ (with $\overline{\overline{\alpha}} = \alpha$ and $\overline{\alpha} \neq \alpha$) causes no difficulties to our theory.

The CCS restriction operator has semantics

$$\frac{P \xrightarrow{x} P'}{P \setminus \alpha \xrightarrow{x} P' \setminus \alpha}(x \notin \{\alpha, \overline{\alpha}\})$$

Since α is not τ, this is trivially CSP-like, and indeed is equivalent to the CSP construct $P \underset{\{\alpha,\overline{\alpha}\}}{\parallel} STOP$.

The CCS parallel operator is much more interesting. It has semantics

$$\frac{P \xrightarrow{x} P'}{P \mid Q \xrightarrow{x} P' \mid Q} \qquad \frac{Q \xrightarrow{x} Q'}{P \mid Q \xrightarrow{x} P \mid Q'} \qquad \frac{P \xrightarrow{\alpha} P' \wedge Q \xrightarrow{\overline{\alpha}} Q'}{P \mid Q \xrightarrow{\tau} P' \mid Q'}$$

This is CSP-like, with arity $(2, \emptyset)$ and one set of transition rules corresponding to each of these three clauses: the first two have $(\{(1,\alpha)\}, \alpha, |, \emptyset, \{(1,1),(2,2)\}, \emptyset)$ and $(\{(2,\alpha)\}, \alpha, |, \emptyset, \{(1,1),(2,2)\}, \emptyset)$ for all $\alpha \in \Sigma_0$, and the final clause is modelled by $(\{(1,\alpha),(2,\overline{\alpha})\}, \tau, |, \emptyset, \{(1,1),(2,2)\}, \emptyset)$ for all $\alpha \in \Sigma_0$.

Note how similar these are to the rules for $\underset{X}{\parallel}$ quoted earlier. The main structural difference is that both sorts of rule apply to *all* visible events, rather than being partitioned by X.

The model produced by the resulting *SOP* process can be simplified a little to the following. Extend Σ_0 by a separate copy $\Sigma_1 = \{\alpha' \mid \alpha \in \Sigma_0\}$. Let *Prime* be, as before, the renaming that maps α to α', and let *DualPrime* map each α to $\overline{\alpha}'$. Then $P \mid Q$ is equivalent to the CSP construct

$$(Prime(P) \underset{\Sigma_1}{\parallel} DualPrime(Q)) \setminus \Sigma_1$$

We can therefore conclude that, except for +, CCS is CSP-like.

[9] The only difference is that CCS allows this definition to be used unconditionally, even on under-defined terms such as $\mu\, p.p$. The translations from CCS to CSP in this section are therefore restricted to the case where all recursions add at least one initial action.

It is possible to simulate the whole of CCS in CSP, but in a slightly more complex way that does not imply compositionality over CSP models or a theory of refinement. An elementary way of doing this is to replace the event τ by a visible analogue (say *tau* as we have seen elsewhere in this paper), and produce models of CCS operators that replace τ by *tau*. Finally, at the outermost level (just as in *SOP*) we would hide *tau*. Thus the model of a closed piece P of CCS syntax would be the CSP term $P' \setminus \{tau\}$, where P' is the syntax of P with all operators replaced by their CSP analogues. In this model the analogue of $+$ would be $\Box$, since *tau* does resolve $\Box$, and the model of parallel would be as above (noting that *tau* is not synchronised) except that the outer hiding $\setminus\ \Sigma_1$ would be replaced by a renaming that sends all members of Σ_1 to *tau*.

This provides a way of calculating the operational semantics of a CCS term using those of CSP, and also an easy method for using CSP tools such as FDR on such terms.

It is not, in fact, necessary to leave *all* the *tau* actions visible as we build up a term, only those that might resolve a $+$ operator for which our present process becomes (directly or indirectly) an argument. A little structural analysis shows that the only relevant *tau*s are those that are the *first* action that our process performs. It follows that if we apply the following CSP-like operator (standing for "Hide Delayed Taus") to any term as we build up P', the final result is not affected:

$$\frac{P \xrightarrow{\tau} P'}{HDT(P) \xrightarrow{\tau} HDT(P')} \qquad \frac{P \xrightarrow{a} P'}{HDT(P) \xrightarrow{a} P \setminus \{tau\}} (a \neq \tau)$$

This is easy to represent in CSP using double renaming of *tau*. The first (τ-promoting) operational clause will never fire in this particular sort of use as the CCS-modelling terms (pre-hiding) never have initial τ actions. This operator might well be useful if one wants to apply CSP model compressions such as those of FDR [19] in a hierarchical way to CCS constructs, as might the simpler observation that it is always safe to hide any *tau* at a level in the syntax above any $+$.

An obvious question then arises: can one model CSP in CCS? The immediate answer to this is "no" since CCS cannot model the multi-way synchronisations permitted by CCS: as soon as two events are synchronised in CCS they are hidden. Of course there may be further interesting questions to ask here about subsets of CSP or extensions of CCS.

6.2 Towards mobility in CSP

The concept of *mobility*, crystallised elegantly in notations like the π-calculus [21, 10], means the ability to pass capabilities that are richer than just data objects along channels. The simplest case of this is where channels (and thus the capability of talking to the process who holds the other end of a channel) can be communicated. In this section and the next we will show how mobility can be brought within the "CSP-like" world.

First, let us recall Hoare's vision of the CSP parallel operator in which all processes come equipped with *alphabets* defining the set of actions that they participate in within parallel compositions. This leads to the binary operator $P\ {}_A\|_B\ Q$ in which the respective alphabets of P and Q are A and B, and by extension the indexed form $\|_{i=1}^{n}\ (P_i, A_i)$ that we have used frequently in this paper. This parallel operator has a natural commutativity when the alphabets follow the processes, and associativity provided the alphabet of any sub-composition is the union of those of its parts: thus

$$(P\ {}_A\|_B\ Q)\ {}_{A \cup B}\|_C\ R = P\ {}_A\|_{B \cup C}\ (Q\ {}_B\|_C\ R)$$

Now imagine that the alphabets change dynamically as the rights to use particular events are transferred from one process to another along special, *rights* channels $s \in RC$. A communication on such a channel will, at least for us, be one of the following:

- An input $\underline{s}.r$: by communicating this event a process gains the right r receives on s.
- An output $\overline{s}.r$: the counterpart, in which the process loses the right.

Here, a *right* is usually either an event $a \in \Sigma_0$ or is the right to input (e.g., $\underline{s}$) or output ($\overline{s}$) along one of the rights channels. We will add one further sort of right below. Here, Σ_0 is the set of ordinary CSP synchronisations and does not contain the communications along rights channels.

Notice that while CSP events in Σ are normally thought of undirected synchronisations between processes, it is natural to think of a rights channel $s \in RC$ to have a direction, being *from* one process and *to* another. This is because we need to know which way a right travels along it.

What we will do below is create an analogue of the indexed parallel operator designed to allow this. We use the indexed parallel operator rather a binary one because the problems of accounting for rights become much easier if we are presented with a closed system rather than an open one: specifically we demand that no rights channel points into or out of the network that we build using our extended parallel. For a particular network $N = \{(P_i, A_i) \mid i \in \{1, \ldots, n\}\}$ we will have a set R of those rights channels that are internal to it. For such channel s, the rights $\overline{s}$ and $\underline{s}$ each belong to exactly one initial alphabet A_i. These properties are preserved as invariants as the network evolves because communication on these channels is point to point and the network is closed.

The alphabets A_i will themselves be composed of events $a \in \Sigma_0$ ("ordinary" communications), and rights of the form $\underline{s}, \overline{s}, s$ for $s \in RC$. The undecorated right s means that the process has the right to *observe* the channel s, which must not be in R. In this initial treatment we will assume, for clarity, that no right of the form $\underline{s}$ or $\overline{s}$ belongs to any A_i for $s \notin R$, and that no right of the form s belongs to more than one A_i.

The visible actions of the resulting language are of one of the forms

- $a \in \Sigma_0$: "ordinary" communications
- $\overline{s}.r$ and $\underline{s}.r$ for $s \in RC$ and $r \in RT = \Sigma \cup \{\overline{s}, \underline{s} \mid s \in RC\}$.
- $s.r$, which is the result of combining $\overline{s}.r$ and $\underline{s}.r$.

If δ is an event of the second form, let δ' denote the event obtained by reversing the sense of the channel (but not the right if that, too is a channel). Thus $(\overline{s}.\overline{t})' = \underline{s}.\overline{t}$.

We will therefore be defining a parallel operator $[\![\!]_{i=1}^{n}(P_i, A_i)$ under the condition that, for each $s \in R$, $\overline{s}$ and $\underline{s}$ each belong to exactly one A_i, and s (the right to observe s) belongs to no A_i. Furthermore, the conditions set out above about rights relating to channels $s \in RC - R$ apply: these guarantee that no rights relating to R can pass in or out of our system, amongst other things.

If A is a set of rights, let $|A|$ denote the set of all actions that a process can perform if it has these rights. For a network whose internal rights channels are R, let $O(R)$ be the set of its "ordinary communications", namely $\bigcup_{i=1}^{n} |A_i| cap \Sigma_0$.

The operational semantics for our operator are as follows. τs are promoted as usual:

$$\frac{P_j \xrightarrow{\tau} P_j'}{[\![\!]_{i=1}^{n}(P_i, A_i) \xrightarrow{\tau} [\![\!]_{i=1}^{n}(P_i' \triangleleft i = j \triangleright P_i, A_i)}$$

The following rule shows that for ordinary events, the usual synchronisation rules apply.

$$\frac{\forall j.x \in |A_j| \Rightarrow P_j \xrightarrow{x} P'_j}{\mathbb{R}^n_{i=1}(P_i, A_i) \xrightarrow{x} \mathbb{R}^n_{i=1}(P'_i \lhd x \in |A_i| \rhd P_i, A_i)}(x \in O(R))$$

Suppose $inps(A) = \{s.r \mid \underline{s} \in A \cap R \wedge r \notin A\}$ and $outs(A) = \{s.r \mid \overline{s} \in A \cap R \wedge r \in A\}$ are respectively the sets of input and output events that make sense for the alphabet A. (One could imagine weakening the constraint expressed here that a process can only input a right it does not already have it.) The rule for a communication over a channel $s \in R$ is then:

$$\frac{P_j \xrightarrow{\overline{s}.r} P'_j \wedge P_k \xrightarrow{\underline{s}.r} P'_k}{\mathbb{R}^n_{i=1}(P_i, A_i) \xrightarrow{s.r} \mathbb{R}^n_{i=1}(Q_i, B_i)}(j \neq k \wedge s.r \in inps(A_k) \cap outs(A_j)) \quad \text{where}$$

$$(Q_i, B_i) = \begin{cases} (P'_j, A_j - \{r\})) & \text{if } i = j \\ (P'_k, A_k \cup \{r\})) & \text{if } i = k \\ (P_i, A_i) & \text{otherwise} \end{cases}$$

Notice that, for $s \in R$, the composition $\mathbb{R}^n_{i=1}(P_i, A_i)$ never communicates externally an event of the form $\overline{s}.r$ or $\underline{s}.r$, merely the undirected event $s.r$.

Whether or not it is thought that this is a reasonable approach to adding a certain sort of mobility to standard CSP, it is a good illustration of the power of our new operational approach to defining CSP-like operators. The result is a family of n-ary operators on processes (all for which are **on**), in which the various alphabets are (as with the ordinary CSP parallel and hiding operations) parameters of the operator. So the changes to the alphabets implied in the last rule means that a slightly different operator is being applied to the successor arguments.

While the above presentation is fairly complex, it is greatly simpler and more intuitive than any definition over traces, failures etc could be. And of course Theoremr̃efth2 guarantees that the new operator does make sense over these models. It is possible to create the above composition in terms of a binary operator, but the "closed world" we have created for ourselves in the network means that in defining the indexed operator we have not had to handle some difficult cases that would have been unavoidable in the case of a binary form.

The approach we have taken here has been to stay as close to the CSP model of concurrency as possible with "ordinary events", while being forced to accept a point-to-point directed channel model for the transfer of rights. There are CSP-based languages, notably occam [5], in which this latter model of communication is adopted for all interaction. Arguably these can be extended to include mobility more naturally than CSP itself, and this has been done in the language occam-π [23].

6.3 The π-calculus

The π calculus [13, 12, 21] builds on the notation of CCS by adding the concepts of name binding and name passing into the language. Like CCS, it does not need process alphabets to define parallel, and therefore the way it expresses mobility is more implicit than the CSP-based approach defined above: since there are no alphabets, there is no need to add to or subtract from these. The theoretical attention passes from being focussed solely on concurrency and interaction, to being the combination of this with the issues raise by the binding and value-passing aspects.

The language of the π-calculus is [21] as follows:

PREFIXES $\quad \pi ::= \overline{x}y \mid x(z) \mid \tau \mid [x = y]\pi$

PROCESSES $\quad P ::= M \mid P|P \mid \boldsymbol{\nu} z\ P \mid !P$

SUMMATIONS $\quad M ::= \mathbf{0} \mid \pi.P \mid M + M'$

Here, $\overline{x}y$ represents the sending of the name y via x (akin to the CSP action $x!y$) and $x(z)$ is a construct binding y that represents the receipt of z over x (akin to $x?z$). Like CCS and unlike CSP it has a construct representing the explicit introduction of τ, and one can guard actions with the assertion that pairs of names are the same.

The process constructs are the same as in CCS except that the infinite replication $!P$ (equivalent to $P \mid !P$) takes the place of recursion. The effect of having summations in a separate syntactic class is to restrict summations to appearing in restricted contexts (similar to the "guarded choice" construct introduced as a precursor to $\Box$ in CSP in both of [4, 15]). Some presentations of the π-calculus omit $+$ is omitted from the core language.

For the denotational semantics we will give later, it is convenient to distinguish syntactically between names in the set *Names* on the one hand, and the variables bound by (z) and $\boldsymbol{\nu}\ z$ on the other.

The theory of the π calculus is complex. This is not because processes can pass names around and use them as channels – akin to CSP processes passing channel names around as in the previous section – but because of the way new names are "created" fresh and distinct from all others. In other words, when two names appear in either nested or separate scopes they are assumed to be different. One has to emphasise the latter because it is possible for a label to be exported by communication *outside* its scope and therefore visible to the outside world.

Naturally, the precise choice of which label is chosen at any $\boldsymbol{\nu}x.P$ is unimportant to the overall semantics of the term: it will behave in exactly the same way as it would have done had any other legitimate label been picked, except when that label is visible externally. And even then all external processes and observers will treat these two names alike. What is therefore important about our mechanism for managing these names is that, to the processes, they all seem distinct, not exactly which names they are. This is, in essence, a somewhat extended and unconventional analogue of α-conversion, more difficult to handle semantically because the names that are chosen internally can be viewed from the outside.

The general approach we will take is that the semantics of any term is in effect a nondeterministic choice over all legitimate assignments of names. But these choices are actually much more straightforward than most nondeterministic choices, because the different options affect only the names that can be seen from the outside, not the shape of the subsequent behaviour. We would expect that such choices of names produce processes that are identical except for the application of renamings induced by permutations on the sets of names.

In giving our semantics below we use forms involving infinite nondeterministic choices over very similar processes, because these are perhaps the most elegant from the perspective of CSP. We will discuss alternatives that might be employed in derived verification algorithms below.

The issue of freshness in the π-calculus causes difficulties in interpreting its operational semantics within the framework we are using in this paper. It may be perfectly legitimate for either of the terms P and Q to create the name x as one that is generates and make visible to the outside world. But is not legitimate for them *both* to do it or else the second would not be creating a *fresh* name. We have to contend with questions such as whether a

name is fixed semantically at the point where it is intuitively decided operationally or at the point where it is first communicated to the outside world.

It follows from this that we may need to assign a particular infinite set of names from which a given term may choose to be assigned to its $\boldsymbol{\nu} v\, P$ bound variables. Therefore some of the terms in our operational semantics will have the form (P, S) for P a π-calculus process and S an infinite subset of *Names* that is disjoint from $fn(P)$. As well as deriving the transitions for terms of this type directly we will also feel free to apply two different operationally defined operators over transition systems to them. One of these is the CCS parallel operator, that we will call $|_{ccs}$ here to distinguish it from the π-calculus syntax $|$. We already know that this is a CSP-like operator. We will later give the semantics of $|$ in terms of $|_{ccs}$ and some careful management of the choices of labels.

Process actions are drawn from $\{\tau\} \cup \{c.n \mid c \in Chan \wedge n \in Names\}$ where $Chan = Names \cup \{\overline{v} \mid v \in Names\}$. The overline operation on channels is its own inverse, as in CCS.

We first deal with terms of the form $\pi.P$. Naturally $(\tau.P, S) \xrightarrow{\tau} (P, S)$ for all (P, S), and

$$\frac{(\pi.P, S) \xrightarrow{x} (Q, S')}{([y = y]\pi.P, S) \xrightarrow{x} (Q, S')}$$

The output and input actions for communicating names are described:

$$\frac{}{(\overline{x}y.P, S) \xrightarrow{\overline{x}.y} (P, S)} \qquad \frac{}{(x(y).P, S) \xrightarrow{x.y} (P[y/x], S - \{y\})}$$

We now consider the other forms of summation M: naturally $\mathbf{0}$ has no actions, and

$$\frac{(M_1, S) \xrightarrow{x} (M_1', S')}{(M_1 + M_2, S) \xrightarrow{x} (M_1', S')}(S \cap fn(M_2) = \emptyset)$$

$$\frac{(M_2, S) \xrightarrow{x} (M_2', S')}{(M_1 + M_2, S) \xrightarrow{x} (M_2', S')}(S \cap fn(M_1) = \emptyset)$$

The side conditions here are just to ensure that the combination $(M_1 + M_2, S)$ is well formed: $fn(M)$ is the set of free names in M, for any process or summation M.

We next consider the binding form $\boldsymbol{\nu} x\ P$ which creates a fresh name for use within P and prevents it being seen outside except in carefully controlled circumstances. Namely, x must be output from P via an event of the form $\overline{y}x$. $(y \neq x)$ *before* x is seen in any other circumstances. This creates a near paradox: imagine the term $\boldsymbol{\nu} x\ y(z).\overline{z}x.\mathbf{0}$ This creates a new name x, inputs a second name z on the free channel y and then outputs x along z. But the environment is completely free to send our process *any* name, including x, along y. If it did so this would violate the principle of freshness. A solution to this, and the one we adopt, is to realise that, up to the point when x is first seen by the outside world, its actual identity does not matter. So we create an operator that, in effect, carries out an α-conversion, but in a semantic rather than syntactic sense.

To achieve this we make the term $(\boldsymbol{\nu} x\ P, S)$ choose not only a value for x from S, but also an infinite set of names that can replace x in "emergencies" like this one. We need this *infinite* set of reserves because they might also get used up as our process progresses. The operator $OF(X, Y, P)$ (with Y being an infinite set of names disjoint from the free names of P and the set of names X) has the effect of protecting the names in X so they are definitely "Output First", even if we have to change their outward names to achieve this. It

also prevents any transition P might try to perform that would show x before it has been properly output, thereby implementing the *restriction* aspect of the $\boldsymbol{\nu} v\ P$ construct.

It is important to realise that since we are affecting only the external appearance of the names X, the way our process behaves inwardly is still based on whatever members of X it has created. We therefore have to change not only how members of X created by P appear to the outside world, but also how any member of X introduced by the outside world appears to P. Thus, each time the outside world uses a member x of X, we not only replace the outward appearance of any x's created by P by some $s \in S$, but we also replace the inward appearance of the external x by s. In other words, from the point where the external x is first seen, P behaves like $P[\![^{x,\,s}/_{s,\,x}]\!]$: the CSP renaming that swaps all occurrences of s and x in what P does. Of course it is important that we use this construction only when the external value x would have been fresh for P had P chosen a different value for the bound variable, and when the bound variable's scope has not been extended beyond P.

For example, consider the term

$$(b(v).\overline{a}v.v(y).[v = y]b(v').\mathbf{0}) \mid (\boldsymbol{\nu} z\ a(t).\overline{t}z.a'(z').[z = z']c(w).\mathbf{0})$$

Suppose the left-hand process P inputs the name x over b from the outside world after the right hand process Q has chosen x as the value of the bound variable z. If this co-incidence were allowed to take effect then the equality guard in P would be true, meaning that the second input on b is enabled. In a world where the choice of z is assumed to be fresh in the strongest sense, however, this should not happen: the result of the equality test should always be false. We can achieve this by the use of our proposed operator OF to the body of the declaration in Q: $OF(x, S, a(t).\overline{t}x.a'(z').[x = z']c(w).\mathbf{0})$ will spot when x is input along a, so the subsequent behaviour will be $(\overline{t}x.a'(z')[x = z']c(w).\mathbf{0})[\![^{x,\,s}/_{s,\,x}]\!]$ for a fresh s. So the equality guard in this process will be true if and only if the outside world communicates s to Q on a'. In other words the overall behaviour will be identical to what would have happened if Q had picked s as the value of its bound variable rather than the clashing x.

We need to give one word of caution about the effect that OF has on process evolutions. If a particular value z is picked for the x in $\boldsymbol{\nu} x\ P$ but is not output from this context for some time, then the still-implicit value might exist down several branches of the execution of P. Down most of these it may not have to be swapped with another s, whereas this may be necessary on one. We will thus have a computation tree of $\boldsymbol{\nu} x\ P$ on which, for different branches, we will have x playing two quite different roles and two different values for the value introduced in this restriction. It seems to the author that something like this, though ugly, is necessary in any semantics where inputs on channels are completely arbitrary. For reasons we will discuss later, this causes few or no problems at all in CSP models.

OF is a CSP-like operator that we can define operationally as follows:

$$\frac{P \xrightarrow{x} P'}{OF(X, Y, P) \xrightarrow{x} OF(X, Y, P')}(ind(x, X \cup Y))$$

where $ind(x, Z)$ means that the transition x uses no name that appears in the set Z of names. Note that $ind(\tau, Z)$ for all Z.

$$\frac{P \xrightarrow{\overline{y}.x} Q}{OF(X, Y, P) \xrightarrow{\overline{y}.x} OF(X - \{x\}, Y, P)}(x \in X \wedge ind(y, X \cup Y))$$

In other words, P may output any member x of X along a channel that is not restricted, and by doing so remove the restriction on x.

$$\frac{P \xrightarrow{y.x'} Q}{OF(X, Y, P) \xrightarrow{y.x} OF((X \cup \{x'\}) - \{x\}, Y - \{x'\}, P[\![^{x'}, {}^{x}/_{x, x'}]\!])}(x \in X \wedge x' \in Y \wedge ind(y, X \cup Y))$$

If $OF(X, Y, P)$ inputs a protected name then the protected name is replaced by any member of S via renaming as in the example above. Note that the externally visible action $y.x$ is translated into $y.x'$ in P since we want P to input the name x' that is fresh to it rather than x which is not. Similarly any other xs apparent in external communications need to appear as x' to P, and when P does choose to output its value x then to the outside world it appears like x': fresh to the outside world, unlike x.

$$\frac{P \xrightarrow{y.x} Q}{OF(X, Y, P) \xrightarrow{y.x} OF(X, Y - \{x\}, P)}(x \in Y \wedge ind(y, X \cup Y))$$

If a value is input that coincides with a reserve value in Y, then it is removed from the reserves.

Note that since there is no rule that allows them, all initial communications of the form $x.y$ and $\overline{x}.y$ for $x \in X \cup Y$ are prevented from happening by this operator.

We can express the operational semantics of $\boldsymbol{\nu} x\ P$ thus:

$$\frac{}{\boldsymbol{\nu} v\ P \Rightarrow OP(\{v\}, Y, P[v/x]), S - (\{v\} \cup Y)}(v \in S \wedge Y \prec S)$$

meaning that term on the left hand side of the arrow $\Rightarrow$ must transform itself into one of the terms on the right hand side that meet the side condition. In CSP we would normally replace $\Rightarrow$ by $\xrightarrow{\tau}$, but this would introduce internal actions one would not expect into the π-calculus.

The notation $Y \prec S$ above means that $Y \subset S$ and both Y and $S - Y$ are infinite. This notation is useful because it sets out that both names Y and the complement $S - Y$ contain sufficient for any length of execution. In this case they respectively provide the reserve names for x and the names that $P[v/x]$ might create.

Similar considerations are needed for the parallel operator $|$: the necessary side condition here is that the name sets associated with the two components are disjoint from each other and from both processes' free variables: if both of P and Q extrude names they have invented, we need to ensure in $P \mid Q$ that they are different.

Another important thing to realise is that, through scope extrusion, the combination $P|Q$ can become the scope of an as-yet unseen name devised by one of them. For the name might have been communicated from P to Q via a hidden communications between them, and not yet revealed to the outside world. The freshness of such names needs to be protected exactly as though they were still within the original $\boldsymbol{\nu} v\ P$. So, if the fresh names invented by P and Q respectively lie within T and U, we need to select a further disjoint infinite set V of events and apply the operator $OF(T \cup U, V, \cdot)$ to the result of running P and Q in parallel.

Therefore, to run $(P|Q, S)$, we select three disjoint infinite subsets T, U, V that partition S, evaluate (P, T) and (Q, U), put them in parallel using the CCS $|$ operator between LTSs (termed $|_{ccs}$ below), and apply OF as described above. We can summarise this in the following rule:

$$\frac{\{T, U, V\} \text{ partition } S, \text{ all infinite}}{(P|Q, S) \Rightarrow OF(T \cup U, V, (P, T) \mid_{ccs} (Q, U))}$$

We prefer to give a rule for recursion rather than the more restricted replication $!P$:

$$\frac{}{(\mu\, p.P, S) \Rightarrow (P[\mu\, p.P/p], S)}(\text{no binding})$$

The *no binding* condition asserts that no free identifier of $\mu\, p.P$ becomes bound at any place where it is substituted. We can ensure this holds by changing the names of P's bound identifiers to be different from its free ones.

If we are computing the value of a recursion with a view to investigating relationships with CSP, then the use of this rule must be restricted to contexts P that make no recursive call till at least one action has occurred. This means that we will have to be careful with replication $!P$. However, it has been observed that it is sufficient for the π-calculus to consider replication of the form $!(x(y).P)$, which is equivalent to the guarded recursion $\mu\, p.x(y).(P \mid p)$.

Denotational semantics We know that the + operator is not CSP-like; but everything else in π-calculus apart from the name handling is essentially something that is in CSP or the CSP-like fraction of CCS. Furthermore, the main tool we used in handling names in our operational semantics was the CSP-like OF.

The problem with + is actually less than in CCS, because the syntax of the π-calculus given earlier restricts its use to the subclass of summations: it is not a "first class operator" over the process type. This suggests that we can give a natural CSP denotational semantics for the process type based on hiding the *tau* events that will be visible in the semantics of summations.

The main consequence, as far as we are concerned, of an operator being CSP-like is that it makes sense to define it over CSP models. Recall that we defined a language to be CSP-like if it had a denotational semantics into every CSP model. In this section we show this is true for the π-calculus, which requires somewhat more analysis than simply being an application of Theorem 2 thanks to the roles of freshness and +.

We choose to give such a semantics very much in the style of Strachey as expounded, for example, in Stoy [22]. That is, we define a semantic function $\mathcal{E}[\![P]\!]$ of process terms that depends on two parameters, an *environment* and a *state*. The first of these captures the bindings of the identifiers in P, and the second captures that which evolves in time and might well be different at a recursive call from the point where the recursion was defined. Thus $\mathcal{E}[\![P]\!]$ is a function $Env \to St \to \mathcal{M}$, where $\mathcal{M}$ is the CSP model selected.

There is a subsidiary semantic function $\mathcal{S}[\![M]\!]$ of the same type, that applies to summations M and which differs by using *tau* to model initial τ guards. They are therefore linked by the equation

$$\mathcal{E}[\![M]\!]\rho\sigma = \mathcal{S}[\![M]\!]\rho\sigma \setminus \{tau\}$$

A given environment $\rho \in Env$ has to assign values to two different sorts of identifiers that might appear in a term P. The first of these are the "variables" v which are either bound by (v) or $\boldsymbol{\nu} v$ or are free in P. It is cleaner, in defining a denotational semantics, to take the trivial syntactic step from simply having a type of "names" to having both names, and variables that denote names, in the language. We will allow either of these two appear free in a process expression, but only have bound variables. For any term P, $fv(P)$ are the free variables of P, and $fn(P)\rho$ consists of the syntactic names appearing within P together with $\{\rho(v) \mid v \in fv(P)\}$. This is the natural extension of the usual definition of the free names function to our world with variables and environments.

The second type of identifiers are those that represent the values of the process variables representing recursive calls. The appropriate values to assign to these are functions $St \to \mathcal{M}$.

A state $\sigma \in St$ will simply be an infinite subset of *Names*, the set of names from which our term P is free to pick from for $\boldsymbol{\nu} v$ constructs. It must, for example, exclude all names that are meaningful in any context that encloses our term P as well as P's own free names.

We can now define our semantic functions. Throughout this definition we will assume that P, ρ and σ are consistent in the sense that (i) no free name of P nor any element of the range of ρ is a member of σ, and (ii) that every free variable of P is in the domain of ρ.

In the syntax we are using, a syntactic object x that is to be interpreted as a name might either be a name or a variable. We will represent its value by x^ρ: this is x if it is a name, and $\rho(x)$ if a variable.

- $\mathcal{S}[\![\mathbf{0}]\!]\rho\sigma = STOP$
- $\mathcal{S}[\![\tau.P]\!]\rho\sigma = tau \to \mathcal{E}[\![P]\!]\rho\sigma$
  This says that $\tau.P$ is equivalent to P in any CSP model.
- $\mathcal{S}[\![[x = y]\pi.P]\!]\rho\sigma = (x^\rho = y^\rho)\&(\mathcal{S}[\![\pi.P]\!]\rho\sigma)$
  In other words, this process behaves like P or $STOP$ depending on whether the guard is true or not.[10]
- $\mathcal{S}[\![\overline{x}y.P]\!]\rho\sigma = \overline{x^\rho}.y^\rho \to \mathcal{E}[\![P]\!]\rho\sigma$
  The output of a name is just an ordinary CSP communication.
- $\mathcal{S}[\![x(y).P]\!]\rho\sigma = x^\rho?z \to \mathcal{E}[\![P]\!]\rho[z/y](\sigma - \{z\})$
  The input of a name is just an ordinary CSP input, where the range of z is the set of values for which $x^\rho.z$ is an event, i.e. *Names*.
- $\mathcal{S}[\![M + M']\!]\rho\sigma = \mathcal{S}[\![M]\!]\rho\sigma \,\Box\, \mathcal{S}[\![M']\!]\rho\sigma$
  As in CCS, this model is accurate when we turn τ into the visible *tau*.
- $\mathcal{E}[\![M]\!]\rho\sigma = \mathcal{S}[\![M]\!]\rho\sigma \setminus \{tau\}$
- $\mathcal{E}[\![P \mid Q]\!]\rho\sigma = \sqcap\{ OF(T \cup U,\, V, \mathcal{E}[\![P]\!]\rho U \mid_{csp} \mathcal{E}[\![Q]\!]\rho V) \mid T, U, V \prec \sigma \text{ disjoint}\})$
  Here $\mid_{csp}$ means a CSP representation of $\mid$ such as that given in Section 6.1.
- $\mathcal{E}[\![\boldsymbol{\nu} v\ P]\!]\rho\sigma = \sqcap\{ OF(\{z\}, S, \mathcal{E}[\![P]\!](\rho[z/v])(\sigma - (\{z\} \cup S))) \mid z \in \sigma \wedge S \prec \sigma - \{z\}\}$ The definition of restriction is as discussed earlier.
- $\mathcal{E}[\![\mu\, p.P]\!]\rho$ is a fixed point of the function mapping a given value $\zeta : St \to \mathcal{M}$ to $\mathcal{E}[\![P]\!]\rho[\zeta/p]$. The nature of the fixed point varies with the CSP model $\mathcal{M}$. For this work we also need the following clause for a simple process identifier:
- $\mathcal{E}[\![p]\!]\rho\sigma = \rho(p)\sigma$.

We present these semantics here without detailed analysis or comparison with established semantics, which must be the topic of subsequent papers. However we make the following comments on them:

- If P is a process expression with no free identifiers (i.e. all free names appear as themselves in the syntax rather than the values of variables, and there are no process identifiers not bound by a recursion), then it seems appropriate to identify the semantics of P with $\mathcal{E}[\![P]\!]\rho_0(Names - fn(P))$, where ρ_0 has an empty domain.
- It seems reasonable to expect that semantics based on CSP models will make less distinctions as a general rule than ones based on bisimulation. An exception to this might be CSP's modelling of divergence.

[10] Recall that $b\&P$ is a CSP abbreviation for *if b then P else STOP*.

- We would expect the essential semantics $\mathcal{E}[\![P]\!]\rho\sigma$ not to change with the consistent state σ. We might formulate this by saying that if $\sigma' \subseteq \sigma$, then $\mathcal{E}[\![P]\!]\rho\sigma'$ consists of precisely those $\mathcal{M}$-behaviours of $\mathcal{E}[\![P]\!]\rho\sigma$ such that all the freshly-generated values by P lie in σ', and that if $g : \sigma \to \sigma$ is a bijection that leaves all non-fresh values (i.e. all input and free values) unchanged, then $b \in \mathcal{E}[\![P]\!]\rho\sigma$ if and only if $g(b) \in \mathcal{E}[\![P]\!]\rho\sigma$.
- We observed earlier that the construction *OF* creates process evaluation trees that are counter-intuitive thanks to a given choice of value for a $\boldsymbol{\nu} v$ only clashing with an external name on one of a number of possible branches. This issue is considerably reduced when we look at the CSP model of a process, since these models only record linear behaviours. However it is not completely eliminated in any standard model that allows one to see actions after a refusal or acceptance. Consider, for example, the refusal testing model in which a process is represented as an alternating series of refusal sets and actions. The process

  $$\boldsymbol{\nu} v\ ((\overline{a}v.\mathbf{0}) \mid (b(z).\mathbf{0}))$$

  will select some value (x, say) for v and then run a pair of processes in parallel. A legitimate execution path is for our process to input on b and then output on a. *OF* will make sure that the value appearing on a will be different from that input on the first step, but there is a problem. Our process could have been observed refusing all events on $\overline{a}$ other than $\overline{a}.x$ before $b.x$ occurred, and then communicating some different $\overline{a}.s$. This is counter-intuitive.
  There is, however, a ready solution to this. There is no way in the π-calculus that a process can input *selectively* along a channel: if it accepts any $x.y$ then it must accept all $x.z$. This will mean that we can use variants of all CSP models in which there are refusal sets and/or ready/acceptance sets. Instead of these events consisting of events they should be convertible to consist of all channels.
  If one were to take this step then all one will have observed before $b.x$ is that the channel $\overline{a}$ was not refused on the previous step. This completely eliminates the sort of paradoxical half-revelation discussed above.
  This should mean, in fact, that CSP models give particularly clean semantics for the π-calculus.
- Since we are using separate semantics functions for processes and summations, the notions of equivalence induced on these two syntactic classes will be different. For example $\tau.\mathbf{0}$ and $\mathbf{0}$ are inequivalent as summations but equivalent as processes. The $\mathcal{S}[\![M]\!]$ semantics on summations is quite closely connected to the operational semantics since it lets us see initial *tau*s, and is deliberately less abstract. It should be thought of as a tool towards creating $\mathcal{E}[\![P]\!]$ rather than being a natural semantics in itself.
- Since CSP models come equipped with a compositional theory of refinement, so will these semantics. As in CSP, refinement will roughly correspond to reduction of nondeterminism. It remains to be seen how useful this will be.
- The way nondeterministic choices are used whenever we have to select a new name or set of names will be vital in maintaining the abstraction of our semantics. Ordinarily speaking, the use in CSP of infinite nondeterministic choices like these restricts one to models in which one does not have to infer infinite behaviours from finite ones (so, for example, it is necessary to add a component of infinite traces to the failures/divergences model). The author conjectures, however, that because all of the the behaviours resulting from different choices in our semantics are isomorphic except for the names of fresh names, it will not be necessary to exclude these models on this account.

– It is usual [21] to specify that the set *Names* is countably infinite. Now that we have the opportunity, by appropriate means, to inspect infinite traces of a process there might be an argument for insisting it is *uncountably* infinite. For as long as it is countable we might well ask how the term $\boldsymbol{\nu} v\ (\overline{x}v.\mathbf{0} \mid (!y(z).\mathbf{0}))$ behaves when the environment systematically outputs every single name in succession to it along y. And it at least raises the question of whether the processes

  $$!(\boldsymbol{\nu} v\ \boldsymbol{\nu} w\ \overline{x}w) \qquad \text{and} \qquad !(\boldsymbol{\nu} w\ \overline{x}w)$$

  are equivalent in infinite traces. At first sight it might appear that the right hand process can systematically output all names one after another, but the one on the left cannot as there must be an infinity of names chosen that are never output. In fact these terms are equal in our semantics because the semantics of $\boldsymbol{\nu} v\ P$ reserves an infinity of names that must not be used by P; but that raises other questions.
  These things would simply not be issues if the set *Names* were uncountable!
  The author believes that if these semantics are developed further there will be strong arguments for *either* adopting an uncountable *Names or* refining the strategies by which names are selected and reserved within a semantics based on countably infinite *Names* so that artificial semantic inequalities between processes based on whether they reveal all names or not on some infinite trace are avoided.
– One way of accomplishing this goal might be to apply some sort of standardised names to the fresh values exported by any process. Given one of the linear behaviours observed in a CSP model semantics such as traces, we could represent that trace, failure etc by labelling the exported names in the order they appear, in a style similar to some approaches to eliminating bound names (and the complexities of α-conversion) to process terms. Thus the first value to appear in a trace might always have label n_1, the second n_2 etc. We would in effect be abstracting a behaviour into something akin to a λ-expression in which each fresh value is thought of a bound variable in the trace. The names of these "bound variables" would be distinct from the set *Names* itself. One of these modified traces etc would have similarities to a head-normal form.
  This would, in a sense, give a more compact representation of a process and would also eliminate the sorts of issue raised in the last paragraph.
  The fact that CSP behaviours are recorded linearly means that this type of transformation of semantics will not create the same conceptual difficulties that it would if the semantics recorded branching behaviours: a single behaviour will not display something that has one name inside the process in different ways. Consider, for example, the process $\boldsymbol{\nu} x\ \boldsymbol{\nu} y.(\overline{z}x.\mathbf{0} + \overline{t}y.\mathbf{0}$.
– It seems likely that most CSP operators could be added to the π-calculus in respect of this method of modelling. It would be necessary to place restrictions and allocate names to different arguments in appropriate ways. For example, we would not want $\overline{x}$ to be renamed to anything other than other output channels; it is probably desirable not to allow the hiding of input communications $x(v)$ since the result would not naturally obey the scoping restrictions of $\boldsymbol{\nu} v$. All renaming, hiding and other event-based objects such as process alphabets should operate only on channel names x or $\overline{x}$ as opposed to events of the forms $x.y$ or $\overline{x}.y$.

7 Conclusions

We have been able to demonstrate that a surprisingly wide range of constructs can be modelled in CSP and therefore in CSP models, even CCS if one is prepared to accept the

two-stage process that this entails. It therefore does not seem too preposterous to hold the view that CSP is a natural "machine code" or semantic notation for modelling event-based concurrent systems.

While, in principle, our methods provide a general route to implementing any suitably finitary CSP-like operator in FDR, the author suspects that the fully general methods described in this paper will be too expensive (for example in alphabet size) for it to be worthwhile developing a general "operator compiler" through the medium of CSP. He believes, however, that the following will be practical:

- As we have done at several points in this paper, for example with CCS |, one could take advantage of the particular shape of an operator to derive a more efficient customised representation in CSP.
- Develop a way of inputting a CSP-like operational semantics directly into FDR so that it can implement them directly. This accords well with the current strategy for developing FDR, but may well not be simple, particularly when it comes to debugging failed refinement checks.

We have shown how easy it is to add new operators into CSP to meet new needs. The author's experience with the mobile parallel operator of Section 6.2 was that it was far harder to choose a reasonably well-behaved operator (i.e. one where the variable alphabets did not create undesirable situations) than it was to define it. Whether the idea of mobility in CSP has any future really depends on whether it makes the language more usable. It would be natural to add a facility that would allow the dynamic generation of processes within a mobile network, not just passing channels around.

We note that all of the CSP-like operators we needed to develop in Section 6 come within the scope of Section 4.1, where there is a constant population of **on** arguments. However, it seems likely to the author that the results of the other steps in our main proof will come in useful in other types of example: the power to turn arguments on and discard them.

It might seem rather tendentious to propose a new modelling style for the π-calculus without studying the result in detail. Our main motivation in giving it here was to illustrate the methods proposed in this paper for defining semantics. The author hopes that it will prove both accurate and interesting, and perhaps open up new applications and proof tools for this language, as well as opening up a new semantic technique to those already expert in the π-calculus. CSP already has a major application in which freshness plays an important role, namely cryptographic protocols [6, 7], and it is noteworthy that this connection has already been exploited in the Spi-calculus [1]. Various methods have been evolved over the years in CSP to bring protocol checks that would naturally need an infinity of fresh values down to finite state, such as identifying "redundant" keys and having a "manager" process that carefully allocates a finite pool of values [18, 9]. It would be interesting to see whether these, or other aspects of Lazić's work on data independence in CSP, might find applications to the π-calculus through our translation. Certainly one would expect that these or other symmetry-exploitation methods would be both required (to eliminate the explosion created by the free nondeterministic choices of names etc) and relatively easy to apply to a verification model based on these semantics. (The approach to the standardised labelling of fresh names set out at the end of the last section is in essence a symmetry-breaking tool.)

It seems certain to the author that if CSP models of the π-calculus that are richer than traces are to prove useful, then the right models to adopt will be those that, as suggested above, are adapted so that the values appearing in acceptance and refusal sets are channels rather than event names.

One way of looking at the main result of the present paper is that it shows that can express any "reasonable" language subject to the rules (a) no copying of processes in flight and (b) no negative premises in operational semantic rules. There seems to be no prospect of weakening the first of these rules while remaining within the spirit of CSP. On the other had there have been several occasions, particularly concerning the modelling of discrete time[11] [8] where people have defined constructs that simultaneously require some negative premises on operational semantics and require one of the richer CSP models. In every case that the author is aware of, in every case where there was a negative premise on a process this included τ actions. This seems essential if we are to stay within the CSP philosophy that $\tau.P$ is equivalent to P. In essence we are allowing the operator to observe the refusal of some set of events by one of its arguments. This is completely consistent with the view that an operator's view of an argument should be similar to the environment's view of a complete process.

A possible topic of future research is to ask whether, by adding some given operator with this type of operational semantics to CSP+ (just as we added Θ_A to be able to handle self-discarding arguments) one could make the language complete for expressing a richer form of CSP-like operational semantics that allows negative premises that always include τ. It may even be the case that there are several different tiers of such semantics, each characterised by its own operator. Indeed, the author conjectures that in cases where an operand whose refusal contributes to some action of OP_λ is always discarded immediately by OP_λ, failures based models will be sufficient to get congruence, whereas in cases where such a process can stay on we will need to use at least refusal testing models.

Appendix: Notation

This paper follows the CSP notation of [15], from which most of the following is taken.

Σ	(Sigma): alphabet of all communications
τ	(tau): the invisible action
Σ^τ	$\Sigma \cup \{\tau\}$
A^*	set of all finite sequences over A
$\langle\rangle$	the empty sequence
$\langle a_1, \ldots, a_n\rangle$	the sequence containing $a_1, \ldots, a_n$ in that order
$s^\frown t$	concatenation of two sequences
$s \leq t$	$(\equiv \exists\, u.s^\frown u = t)$ prefix order

CSP Processes:

[11] The usual condition there is that the time action *tock* cannot happen when there is a τ action available.

$\mu\, p.P$	recursion
$a \rightarrow P$	prefixing
$?x : A \rightarrow P$	prefix choice
$P \,\Box\, Q$	external choice
$P \sqcap Q, \quad \sqcap S$	nondeterministic choice
$P \underset{X}{\parallel} Q$	generalised parallel
$P \setminus X$	hiding
$P[\![R]\!]$	renaming (relational)
$P[\![^{a,\, b,\, \cdots}/_{x,\, y,\, \ldots}]\!]R$	renaming (by substitution)
$P \triangleright Q$	"time-out" operator (sliding choice)
$P \,\triangle\, Q$	interrupt
$P \,\Theta_A\, Q$	exception throwing
$P \lhd b \rhd Q$	conditional choice: *if* b *then* P *else* Q
$P[x/y]$	substitution (for a free identifier x)
$P \xrightarrow{a} Q$	$(a \in \Sigma \cup \{\tau\})$ single action transition in an LTS
$P \Rightarrow Q$	re-writing in operational semantics (no LTS action)
$\mathcal{T}$	traces model
$\mathcal{N}$	failures/divergences model (divergence strict)

References

1. M. Abadi and A.D. Gordon, *A calculus for cryptographic protocols: the spi calculus*, Proceedings of the 4th ACM conference on Computer and communications security, 1997.
2. 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.
3. He Jifeng and C.A.R. Hoare, *CCS is a retract of CCS*, Unifying Theories of Programming Symposium, Springer LNCS 4010, 2006.
4. C.A.R. Hoare, *Communicating sequential processes*, Prentice Hall, 1985.
5. inmos Ltd, *The occam 2 reference manual*, Prentice-Hall, 1988.
6. G. Lowe, *Breaking and Fixing the Needham-Schroeder Public-Key Protocol using FDR*, Proceedings of TACAS 1996.
7. G. Lowe, *Casper: A compiler for the analysis of security protocols*, Journal of Computer Security **6**, pp53-84, 1998.
8. G. Lowe and J. Ouaknine, *On timed models and full abstraction* Proceedings on MFPS XXI, 2005.
9. Eldar Kleiner, *A web services security study using Casper and FDR*, Oxford University DPhil thesis, forthcoming 2008.
10. R. Milner, *A calculus of communicating systems,* LNCS 92, 1980
11. R. Milner, *Commmunication and Concurrency*, Prentice-Hall, 1989
12. R. Milner, *Communicating and Mobile Systems: The Pi Calculus*, CUP, 1999
13. R Milner, J Parrow and D Walker, *A calculus of mobile systems, Parts I/II*, Information and Computation, 1992
14. A.W. Roscoe, *A mathematical theory of communicating processes*a, Oxford University DPhil Thesis, 1982, `/web.comlab.ox.ac.uk/people/Bill.Roscoe/publications/2.pdf`
15. A.W. Roscoe, *The theory and practice of concurrency*, Prentice-Hall International, 1998. Updated version available via `web.comlab.ox.ac.uk/oucl/work/bill.roscoe/publications/68b.pdf`
16. A.W. Roscoe, *Revivals, stuckness and the hierarchy of CSP models*, Submitted for publication. Available at `http://www.comlab.ox.ac.uk/people/bill.roscoe/publications/105.pdf`.
17. A.W. Roscoe, *The three platonic models of divergence-strict CSP* Proceedings of ICTAC 2008, `http://web.comlab.ox.ac.uk//files/283/divstrplato.pdf`
18. A.W. Roscoe and P.J. Broadfoot, *Proving security protocols with model checkers by data independence techniques*, Journal of Computer Security **7**, pp147-190, 1999.

19. A. W. Roscoe, P.H.B. Gardiner, M.H. Goldsmith, J.R. Hulance, D.M.Jackson and J.B. Scattergood, *Hierarchical compression for model-checking CSP, or How to check 10$\hat{2}$0 dining philosophers for deadlock*, In Proceedings of TACAS 1995 LNCS 1019.
20. A.W. Roscoe and Zhenzhong Wu, Verifying Statemate Statecharts Using CSP and FDR In Proceedings of ICFEM 2006.
21. D. Sangiorgi and D, Walker *The π-calculus: A theory of mobile processes*, CUP, 2001.
22. J.E. Stoy, *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*, MIT Press, 1977.
23. P.H. Welch and F.R.M. Barnes, *A CSP model for mobile processes*, Proc CPA 2008 (IOS Press, 2008).