CSP with Hierarchical State

Robert Colvin and Ian J. Hayes

The University of Queensland,
ARC Centre for Complex Systems,
School of Information Technology and Electrical Engineering,
Brisbane, Australia

Abstract. The process algebra CSP is designed for specifying interactions between concurrent systems. In CSP, and related languages, concurrent processes synchronise on common events, while the internal operations of the individual processes are treated abstractly. In some contexts, however, such as when modelling systems of systems, it is desirable to model both interprocess communications as well as the internal operations of individual processes. At the implementation level, shared state is often the method of communication between processes, and tests and updates of local state are used to implement internal operations. In this paper we propose an extension of the CSP language which maintains CSP's core elegance in specifying process synchronisation, while also allowing state-based behaviour. State is treated *hierarchically*, allowing (nested) declarations of local and shared variables. The state can be accessed and modified using a refinement calculus-style *specification command*, which may be optionally paired with event synchronisation. The semantics of the extended language, preserves the original CSP rules. The approach we present is novel in that state is part of the process, rather than a meta-level construct appearing only in the rules.

1 Introduction

The process algebra CSP [7] is a language designed for specifying concurrent systems in which processes interact by synchronising and exchanging information through a set of common events. Because the focus is on interactions, the internal operations of a process are treated abstractly. In this paper we give an extension of CSP to include a construct for declaring state, and a general construct for testing and updating the state. The state can be declared local to a single sequential process, or shared between concurrent processes. The integration of state-based constructs with CSP enables the internal operations of a process to be specified in a familiar, imperative programming style, and modelling of shared-state systems. The extension is designed so that CSP's core elegance in specifying interprocess communication is maintained, and therefore obeys the following constraints: it is "lightweight", in that it includes only a few straightforward language extensions which do not affect existing constructs (syntactic preservation), and therefore remains faithful to the original style of CSP; all existing CSP operational laws remain valid (semantic preservation); it

M. Leuschel and H. Wehrheim (Eds.): IFM 2009, LNCS 5423, pp. 118–135, 2009.

allows both local state as well as shared state in a hierarchical way; and it allows combining state operations with a single synchronisation.

In Sect. 2 we review CSP and present the operational semantics of a subset of the language. In Sect. 3 we extend CSP with a construct for declaring state, and commands for testing and updating state. In Sect. 4 we generalise tests and updates to any relationship between pre- and post-states, and allow them to be combined with event synchronisation in a single atomic action. Related work is discussed in Sect. 5 and we conclude in Sect. 6.

2 Review of CSP

2.1 Syntax

CSP is a process algebra which allows concurrent processes to communicate synchronously via shared events. We base our understanding of CSP on the book by Hoare [7], and present its meaning via an operational semantics in the style of Roscoe [11] and Schneider [12].

Processes interact via a set of events, *Event*. In addition, there are two special events, τ, representing an *internal* (unobservable) event, and $\checkmark$, representing successful termination. A subset of the syntax of a process in CSP is summarised below.

$$P ::= (a \rightarrow P) \mid (P_1 \sqcap P_2) \mid (P_1 \,[\!]\, P_2) \mid (P_1 \parallel P_2) \mid (P \backslash A) \mid (P_1 \,;\, P_2) \mid SKIP_A \mid STOP_A \tag{1}$$

An *event prefix* process $a \rightarrow P$, where $a \in Event$, is one that synchronises on event a before behaving as process P. An *internal choice* between P and Q, written $P \sqcap Q$, nondeterministically chooses between P and Q, without reference to a particular event. An *external choice* between processes P and Q is given by $P \,[\!]\, Q$. Whichever process is the first to perform a synchronisation event with the environment becomes active. Concurrency is written by $P \parallel Q$, which states that the two processes operate in parallel, synchronising on shared events and interleaving non-shared events. A set of events $A \subseteq Event$ within P may be "hidden", written $P \backslash A$, so that any events in A are not visible externally to P (these become *internal* steps of $P \backslash A$). A sequential composition $P ; Q$ behaves as P until P terminates, after which it behaves as Q. The process $SKIP_A$ has only one possible behaviour, which is to terminate successfully and take no further action. The process $STOP_A$ has no behaviour – it may never synchronise or take any other action. Both $SKIP_A$ and $STOP_A$ are parameterised by a set of events A, which forms their *alphabet*, described below. In general, processes may also be parameterised by values, examples of which are given below.

Events can contain values, e.g., $c.v$, where c is a *channel* name and v is a value. Channels are used for passing information from one process to another. By convention, the sending process $c.v \rightarrow P$ is written $c!v \rightarrow P$, and the receiving process is written $c?x : T \rightarrow P(x)$, which represents a process that engages in any event $c.v$ for $v \in T$, then behaves as $P(x)$ (a process dependent on x). In

this paper, to avoid distractions associated with type systems, we assume all values are of a universal type Val, and hence will omit the type qualifier T.

CSP includes a range of other operators, but for the purposes of this paper we take the above subset as core. As an example, consider the specification of a queue in CSP.

$$\begin{aligned} &Qu(\langle\rangle) \mathrel{\widehat=} enq?x \rightarrow Qu(\langle x\rangle) \\ &Qu(\langle y\rangle \frown q) \mathrel{\widehat=} (enq?x \rightarrow Qu(\langle y\rangle \frown q \frown \langle x\rangle)) \,[\!]\, (deq!y \rightarrow Qu(q)) \end{aligned} \tag{2}$$

The state of the queue is maintained as a parameter to the process. Values are enqueued via channel enq and dequeued on deq. We have followed Schneider's [12] style of separating the empty and non-empty cases of the queue parameter. This is required because the deq channel is not available if the queue is empty.

The set of events a process P may engage in is given by its alphabet, $\alpha(P) \subseteq Event \cup \{\checkmark\}$. The alphabet of a process must satisfy the equations below.

$$\begin{aligned} \alpha(a \rightarrow P) &= \alpha(P) \text{ where } a \in \alpha(P) & \alpha(P \backslash A) &= \alpha(P) \setminus A \\ \alpha(P \sqcap Q) &= \alpha(P) = \alpha(Q) & \alpha(P \,;\, Q) &= \alpha(P) \cup \alpha(Q) \\ \alpha(P \,[\!]\, Q) &= \alpha(P) = \alpha(Q) & \alpha(SKIP_A) &= A \cup \{\checkmark\} \\ \alpha(P \parallel Q) &= \alpha(P) \cup \alpha(Q) & \alpha(STOP_A) &= A \end{aligned} \tag{3}$$

Unless otherwise specified, we assume the alphabet of a process is the minimum required to satisfy the above rules. For instance, a non-empty queue can engage in all enq and deq events, while an empty queue can immediately engage in enq events, followed by all events in the alphabet of a non-empty queue. Define $c.Val \mathrel{\widehat=} \{v : Val \bullet c.v\}$, i.e., $c.Val$ is the set of events formed from channel c and a value. Then for any sequence of values, q, the definition $\alpha(Qu(q)) = enq.Val \cup deq.Val$ is valid because it satisfies the equations given in (3).

As a more complex process example, consider a system which is formed from two processes, S and R, which communicate via a buffer implemented as a queue (process Qu). S generates values for the queue and R reads those values and performs some actions on them. Assume the existence of a process $Produce$ which generates a value x. Process S repeatedly starts process $Produce(x)$ and puts x on the queue. Process R reads value y from the queue and then performs some actions via process $Consume(y)$. The whole system, Sys, is formed by running S and R in parallel with $Qu(\langle\rangle)$, and hiding the communication events.

$$\begin{aligned} S &\mathrel{\widehat=} (Produce(x) \,;\, (enq!x \rightarrow S)) \\ R &\mathrel{\widehat=} deq?y \rightarrow (Consume(y) \,;\, R) \\ Sys &\mathrel{\widehat=} (S \parallel R \parallel Qu(\langle\rangle)) \backslash (enq.Val \cup deq.Val) \end{aligned} \tag{4}$$

Process Sys hides all communication on channels enq and deq. Processes external to Sys will not see any of the events associated with enq or deq – all internal communication will appear as a step in τ. (Sys is not a recursive process and hence may declare the hiding internally.)

$$(a \rightarrow P) \xrightarrow{a} P \quad (5)$$

$$(P \sqcap Q) \xrightarrow{\tau} P \quad \text{and similarly for } Q. \quad (6)$$

$$\frac{P \xrightarrow{a} P' \quad Q \xrightarrow{a} Q'}{P \parallel Q \xrightarrow{a} P' \parallel Q'} \quad (7)$$

$$\frac{P \xrightarrow{\mu} P' \quad \mu \notin \alpha(Q)}{P \parallel Q \xrightarrow{\mu} P' \parallel Q} \quad \text{and similarly for } Q. \quad (8)$$

$$SKIP_A \xrightarrow{\checkmark} STOP_A \quad (9)$$

$$\frac{P \xrightarrow{\tau} P'}{P \parallel Q \xrightarrow{\tau} P' \parallel Q} \qquad \frac{P \xrightarrow{a} P'}{P \parallel Q \xrightarrow{a} P'} \quad \text{and similarly for } Q. \quad (10)$$

$$\frac{P \xrightarrow{a} P' \quad a \in A}{P \backslash A \xrightarrow{\tau} P' \backslash A} \qquad \frac{P \xrightarrow{\mu} P' \quad \mu \notin A}{P \backslash A \xrightarrow{\mu} P' \backslash A} \quad (11)$$

$$\frac{P \xrightarrow{\mu} P' \quad \mu \neq \checkmark}{P \,;\, Q \xrightarrow{\mu} P' \,;\, Q} \qquad \frac{P \xrightarrow{\checkmark} P'}{P \,;\, Q \xrightarrow{\tau} Q} \quad (12)$$

Fig. 1. Rules for CSP

2.2 Operational Semantics

The operational semantics for CSP operators is given in Fig. 1, and follows the style of Schneider [12]. Internal transitions are labelled with the internal event τ, and unless otherwise stated, transitions labelled with a are valid for $a \in \mathit{Event} \cup \{\checkmark\}$. Transitions labelled with μ apply for $\mu \in \mathit{Event} \cup \{\checkmark, \tau\}$.

Rule (5) states that a prefixed process $a \rightarrow P$ may take a step in a then behave as P. (Recall that a prefix a is a member of *Event*, i.e., cannot be τ or $\checkmark$.) Rule (6) states that an internal choice between P and Q may evolve to either process, without any visible action. Rule (7) states that P and Q may synchronise on event a if both are able to do so. Rule (8) states that P may evolve by interleaving an event μ that is not in the alphabet of Q. Rule (9) states that the special process $SKIP_A$ transitions in the $\checkmark$ event and then takes no further action. There are no rules for $STOP_A$ – it cannot engage in any actions. We follow the convention of omitting the alphabet subscript on $SKIP_A$ and $STOP_A$ when it is clear from context. Rule (10) states that P in an external choice may take an internal step without affecting the choice itself. Alternatively, if P may take a step with event a then the choice may be made in P's favour. Symmetric rules holds for Q. Rule (11) states that if P may take a step in a and a is hidden, then $P \backslash A$ may take an internal step. Alternatively, $P \backslash A$ may take a step in a if a is not hidden. Rule (12) states that until P terminates, $P \,;\, Q$ behaves as P, after which it behaves as Q.

As an example, an initially empty Qu process can take the following steps.

$$Qu(\langle\rangle) \xrightarrow{enq.v} Qu(\langle v\rangle) \xrightarrow{enq.w} Qu(\langle v, w\rangle) \xrightarrow{deq.v} Qu(\langle w\rangle) \xrightarrow{deq.w} Qu(\langle\rangle)$$

These steps are justified by Rules (5) and (10). If channels *enq* and *deq* were hidden, each transition label above would be τ (Rule (11)). Without the hiding, the queue process could synchronise and exchange values with a concurrent process which is listening to the *enq* and *deq* events via Rule (7), such as processes S and R in (4).

3 CSP Extended with State

We extend the CSP language with operators for testing and updating state, and call the new language CSP_σ. We assume the existence of a set of variable identifiers, *Var*, a set of values of variables, *Val*, and define a *state* as a partial mapping from variables to values, $state \mathrel{\widehat=} Var \pfun Val$. We use σ to denote such partial states; on occasion we refer to a state which is total on *Var*, and we denote such states by Σ.

3.1 Syntax

Three new operators are added to the language, as shown below, where σ is a state, g is a predicate, x is a variable, and E is an expression[1].

$$P ::= (\mathbf{st}\ \sigma \bullet P) \mid ([g] \rightarrow P) \mid ((x \coloneqq E) \rightarrow P) \mid \ldots \tag{13}$$

where '...' includes the operators for CSP from (1). Guards and updates do not appear in process event alphabets.

A local state process $(\mathbf{st}\ \sigma \bullet P)$ defines the variables in the domain of σ to be local to P, with their value in σ giving their initial value. Updates to and accesses of variables in the domain of σ are hidden from external observers, and hence, the local state hides variables in the same way that events in A are hidden by $P \setminus A$. Local states may be declared hierarchically (nested), e.g., $(\mathbf{st}\ \sigma_1 \bullet (\mathbf{st}\ \sigma_2 \bullet P))$, where expressions in P may contain free variables which are in the domain of σ_1 or σ_2. If a variable x appears in both σ_1 and σ_2, the value for x in σ_2 overrides the value in σ_1. We call the combination of all local states for some process its *context*.

A *guard prefix* process $[g] \rightarrow P$ is blocked from proceeding until predicate g is evaluated to true in the current context, after which it behaves as P. An *update prefix* process $(x \coloneqq E) \rightarrow P$ updates the variable x to the expression E evaluated in the current context, then behaves as P. We use the term *action* to cover everything that may appear on the left-hand side of a prefix, i.e., events, guards and updates.

As an example, consider the (at this stage flawed) specification of a queue in CSP_σ. We define a process Q which handles messages along channels *enq* and *deq* as before, explicitly updating queue variable q. The variable q is declared locally to process Qu.

$$\begin{aligned} Qu &\mathrel{\widehat=} (\mathbf{st}\ \{q \mapsto \langle\rangle\} \bullet Q) \\ Q &\mathrel{\widehat=} \begin{array}{l} enq?x \rightarrow (q \coloneqq q \frown \langle x \rangle) \rightarrow Q \\ [\!] \ [q \neq \langle\rangle] \rightarrow deq!head(q) \rightarrow (q \coloneqq tail(q)) \rightarrow Q \end{array} \end{aligned} \tag{14}$$

After an $enq?x$ event, q is explicitly updated and the process repeats. If q is nonempty Q may participate in *deq* events. However, once the second branch is chosen, Q will refuse *enq* events, which may lead to deadlock if the environment

[1] To avoid distractions we assume all expressions are well defined.

is not offering *deq*. The desired behaviour is that the event *deq* may occur, and the second branch selected, only if $q \neq \langle\rangle$. To achieve this we must be able to atomically combine guards with events; this will be explored in Sect. 4. Processes S, R and Sys from (4) may be defined similarly in CSP_σ, except the Qu process does not need a parameter.

Computation sequences can be specified in CSP_σ, as shown by the following process $Sum(X)$ which calculates the sum of the natural numbers up to X using local variables s and i, and then writes the result to non-local variable x.

$$\begin{aligned} Sum(X) &\mathrel{\widehat{=}} (\mathbf{st}\ \{i \mapsto 1, s \mapsto 0\} \bullet S(X)) \\ S(X) &\mathrel{\widehat{=}} \begin{array}{l} [i \leq X] \rightarrow (s := s + i) \rightarrow (i := i + 1) \rightarrow S(X) \\ [\!] \ [i > X] \rightarrow (x := s) \rightarrow SKIP \end{array} \end{aligned} \tag{15}$$

To an external observer, all of the steps prior to the final copy to x are internal since they test and update only the local variables i and s.

3.2 Operational Semantics

Our novel approach to defining the operational semantics of guards and updates is to introduce them as transition labels which are "hidden" by the closest declaring state, in the same way that events may be hidden in CSP. However, because states may only hide some of the free variables referenced by a guard or update, such states only partially hide those transitions. For example, consider the following transitions of process P prefixed by a guard, within a state which maps i to 1. The local state $\{i \mapsto 1\}$ 'hides' i from external observers.

$$(\mathbf{st}\ \{i \mapsto 1\} \bullet [i \leq 5] \rightarrow P) \xrightarrow{\top} (\mathbf{st}\ \{i \mapsto 1\} \bullet P) \tag{16}$$

$$(\mathbf{st}\ \{i \mapsto 1\} \bullet [i \leq x] \rightarrow P) \xrightarrow{[1 \leq x]} (\mathbf{st}\ \{i \mapsto 1\} \bullet P) \tag{17}$$

$$(\mathbf{st}\ \{i \mapsto 1\} \bullet [y \leq x] \rightarrow P) \xrightarrow{[y \leq x]} (\mathbf{st}\ \{i \mapsto 1\} \bullet P) \tag{18}$$

In (16) the transition label is $\top$, which plays a similar role to τ. The guard trivially evaluates to true in the local state, so to an external observer some internal step is taken. In (17) the guard accesses non-local variable x. The externally observable behaviour of this process is that it will evolve to P if $x \geq 1$. The predicate has been partially instantiated according to the local state. In (18) the local state has no effect on the guard: its progress is dependent on non-local variables and hence is externally visible (via the transition label). A process $(\mathbf{st}\ \{i \mapsto 1\} \bullet [i > 5] \rightarrow P)$ cannot transition at all since the guard does not hold in the local context.

Now consider the following transitions of process P prefixed by an update of variable s.

$$(\mathbf{st}\ \{i \mapsto 1\} \bullet s := 0 \rightarrow P) \xrightarrow{s := 0} (\mathbf{st}\ \{i \mapsto 1\} \bullet P) \tag{19}$$

$$(\mathbf{st}\ \{s \mapsto 1\} \bullet s := 0 \rightarrow P) \xrightarrow{\top} (\mathbf{st}\ \{s \mapsto 0\} \bullet P) \tag{20}$$

$$(\mathbf{st}\ \{i \mapsto 1\} \bullet s := i \rightarrow P) \xrightarrow{s := 1} (\mathbf{st}\ \{i \mapsto 1\} \bullet P) \tag{21}$$

Transition (19) describes an update to a non-local variable, in which the update expression is independent of the local state. Transition (20) describes an update of a local variable. The process evolves to P with s updated locally to 0. This is an internal transition and hence labelled with $\top$. In (21) the local context does not include s, but does include a variable in the update expression. Since i is mapped to 1 locally, to an external observer the process appears as an update of s to 1. We consider more complex update examples below.

Before giving the formal rules for state-based constructs we define some notation. For an expression E and state σ, $E[\sigma]$ represents E with its free variables that are in the domain of σ replaced by their value in σ. For instance, if E is the boolean expression $(x = y + 1)$ and σ is $\{x \mapsto 5\}$, $E[\sigma]$ is $(5 = y + 1)$. If E is a predicate, as in the above example, then $[\![E]\!]$ is true if and only if E holds for all values of all its free variables, i.e., $(\forall\, \Sigma \bullet E[\Sigma])$. (Recall that Σ is a state total on Var.) We write $\mathbf{sat}([g])$ if g is satisfiable for *some* values of its free variables, i.e., $(\exists\, \Sigma \bullet g[\Sigma])$. Note that $\mathbf{sat}([g]) = \neg[\![\neg g]\!]$.

As foreshadowed, we extend the set of possible transition labels to include $SCmd$, which contains guards and updates. A transition $P \xrightarrow{[g]} P'$ says that P can evolve to P' if the predicate g is true, and a transition $P \xrightarrow{x := E} P'$ says that P evolves P' and has the effect of updating x to E. The transition $\top$ is a member of $SCmd$: it represents a transition in guard $[g]$ where $[\![g]\!]$. It is the state-based equivalent of τ, but note that $\top \in SCmd$, whereas $\tau \notin Event$. This makes the definition of the rules more compact, since a transition which is allowable in every state ($\top$) is just a special case. By abuse of notation we treat $\top$ as a single entity, although it in fact represents a set of transitions (e.g., $[true], [1 > 0], [x = x]$, etc.).

If we allow μ to also range over $SCmd$ actions, then all of the rules in Fig. 1 still hold. The only construct from (1) which needs an additional rule to handle $SCmd$ transitions is external choice – see Rule (23), which is the state-based equivalent of Rule (10).

Fig. 2 contains transition rules for the extended set of constructs and transition labels. Rule (22) defines the transitions for guards and updates in a similar manner to event prefixing (Rule (5)). Rule (24) states that $(\mathbf{st}\ \sigma \bullet P)$ transitions in $\tau, \checkmark$ or event a if P does. We allow event a to reference state variables (if a represents the passing of information along a channel), therefore a must be (partially) instantiated by the local state. For instance,

$$(\mathbf{st}\ \{x \mapsto 1\} \bullet c!x \to P) \xrightarrow{c.1} (\mathbf{st}\ \{x \mapsto 1\} \bullet P)$$

Rule (25) states that $(\mathbf{st}\ \sigma \bullet P)$ may transition in guard $[g[\sigma]]$ if P may take a transition in $[g]$, and $[g[\sigma]]$ is satisfiable. This proviso ensures that guards that cannot evaluate to true cannot transition. The rule may be compared to Rule (11), in the sense that variables that occur in the domain of σ are replaced in g by their local value in σ. If g only contains free variables that occur in the domain of σ, then $g[\sigma]$ may be evaluated locally: it will either evaluate to true ($[\![g[\sigma]]\!] = true$), in which case the transition can always occur, or it will evaluate to false, and no transition is possible (since $\mathbf{sat}([g[\sigma]])$ will not hold).

$$([g] \to P) \xrightarrow{[g]} P \qquad ((x \coloneqq E) \to P) \xrightarrow{x \coloneqq E} P \tag{22}$$

$$\frac{P \xrightarrow{\top} P'}{P \parallel Q \xrightarrow{\top} P' \parallel Q} \qquad \frac{P \xrightarrow{s} P' \quad s \in SCmd \setminus \{\top\}}{P \parallel Q \xrightarrow{s} P'} \tag{23}$$

and similarly for Q.

$$\frac{P \xrightarrow{\mu} P'}{(\mathbf{st}\ \sigma \bullet P) \xrightarrow{\mu[\sigma]} (\mathbf{st}\ \sigma \bullet P')} \tag{24}$$

$$\frac{P \xrightarrow{[g]} P' \quad \mathbf{sat}([g[\sigma]])}{(\mathbf{st}\ \sigma \bullet P) \xrightarrow{[g[\sigma]]} (\mathbf{st}\ \sigma \bullet P')} \tag{25}$$

$$\frac{P \xrightarrow{x \coloneqq E} P' \quad x \notin \operatorname{dom} \sigma}{(\mathbf{st}\ \sigma \bullet P) \xrightarrow{x \coloneqq E[\sigma]} (\mathbf{st}\ \sigma \bullet P')} \tag{26}$$

$$\frac{P \xrightarrow{x \coloneqq E} P' \quad x \in \operatorname{dom} \sigma \quad v \in Val \quad \mathbf{sat}([E[\sigma] = v])}{(\mathbf{st}\ \sigma \bullet P) \xrightarrow{[E[\sigma]=v]} (\mathbf{st}\ \sigma \oplus \{x \mapsto v\} \bullet P')} \tag{27}$$

Fig. 2. Rules for guards and updates

The example transitions (16)–(18) may all be justified by applying Rule (25) and Rule (22).

Rule (26) states $(\mathbf{st}\ \sigma \bullet P)$ may transition in "$x \coloneqq E[\sigma]$" if P may transition in "$x \coloneqq E$" and x is not local to σ. In this case, the expression E is partially instantiated with respect to σ, and the local state is not affected. Examples of this were given in (19) and (21).

Rule (27) captures the case where the updated variable x is local to P. This case is complicated by the possibility that the value of E may not be determined solely by σ, that is, when E contains variables not in the domain of σ. The rule therefore describes many possible transitions, one for each possible value of v such that $(E[\sigma] = v)$ is satisfiable. In each such transition, the local state is updated so that x is mapped to v. Importantly, the transition label $[E[\sigma] = v]$ below the line is a guard, whereas above the line the label $x \coloneqq E$ is an update. The labelling ensures that the value v chosen for x is consistent with the context. The updated state is described notationally by $\sigma \oplus \{x \mapsto v\}$; more generally, for functions f and g, f overridden by g, $f \oplus g$, is a function which returns $g(x)$ for elements in the domain of g, and $f(x)$ otherwise.

Transition (20) given earlier is a simple example of the application of (27) where we make the obvious choice of 0 for v, since $E[\sigma]$ evaluates to 0, and therefore $[E[\sigma] = v] = [0 = 0] = \top$. Given below is a set of transitions for the more complex case where the updated variable is local but the expression E is not.

$$(\mathbf{st}\ \{s \mapsto 1\} \bullet s \coloneqq s + i \to P) \xrightarrow{[i=0]} (\mathbf{st}\ \{s \mapsto 1\} \bullet P) \tag{28}$$

$$(\mathbf{st}\ \{s \mapsto 1\} \bullet s \coloneqq s + i \to P) \xrightarrow{[i=1]} (\mathbf{st}\ \{s \mapsto 2\} \bullet P) \tag{29}$$

$$(\mathbf{st}\ \{s \mapsto 1\} \bullet s \coloneqq s + i \to P) \xrightarrow{[i=2]} (\mathbf{st}\ \{s \mapsto 3\} \bullet P) \tag{30}$$

In these cases we cannot locally determine the value to which s must be updated, since the update expression accesses non-local variable i. Locally, therefore, there are many possible transitions, one for each $v \in Val$ to which s can be updated (we have shown only the transitions for $v = 1, v = 2, v = 3$). However, in practice, only one transition will be possible for a given context. In this case, that will be the transition in which v has the value of $1 + i$ in context.

For instance, the process can evolve to P with $s = 2$ in the local state only if $i = 1$ in the context (29). Hence, consider an outer context of P in which i has the value 1.

$$(\mathbf{st}\ \{i \mapsto 1\} \bullet (\mathbf{st}\ \{s \mapsto 1\} \bullet s \coloneqq s + i \rightarrow P)) \xrightarrow{\top} (\mathbf{st}\ \{i \mapsto 1\} \bullet (\mathbf{st}\ \{s \mapsto 2\} \bullet P))$$

The assignment $s \coloneqq s + i$ is completely determined by the context provided by the outer state $\{i \mapsto 1\}$, and hence always transitions (in $\top$). The effect is to update the value of s within the inner state. By Rule (25), transitions (28) and (30) are not possible when the outer context determines $i = 1$, since, for example, $\mathbf{sat}([(i = 0)[\{i \mapsto 1\}]])$ does not hold.

Below is the execution of program $Sum(2)$ from (15).

$$\begin{array}{ll}
 & (\mathbf{st}\ \{i \mapsto 1, s \mapsto 0\} \bullet S(2)) \\
\xrightarrow{\top} & (\mathbf{st}\ \{i \mapsto 1, s \mapsto 0\} \bullet \\
 & \quad (s \coloneqq s + i) \rightarrow (i \coloneqq i + 1) \rightarrow S(2)) \qquad \textit{Rules}\ (22), (23), (25) \\
\xrightarrow{\top} & (\mathbf{st}\ \{i \mapsto 1, s \mapsto 1\} \bullet (i \coloneqq i + 1) \rightarrow S(2)) \qquad \textit{Rules}\ (22), (27) \\
\xrightarrow{\top} & (\mathbf{st}\ \{i \mapsto 2, s \mapsto 1\} \bullet S(2)) \qquad \textit{Rules}\ (22), (27) \\
\xrightarrow{\top} & (\mathbf{st}\ \{i \mapsto 2, s \mapsto 1\} \bullet \\
 & \quad (s \coloneqq s + i) \rightarrow (i \coloneqq i + 1) \rightarrow S(2)) \qquad \textit{Rules}\ (22), (23), (25) \\
\xRightarrow{\top} & (\mathbf{st}\ \{i \mapsto 3, s \mapsto 3\} \bullet S(2)) \qquad \textit{Rules}\ (22), (27)\ (\times 2) \\
\xrightarrow{\top} & (\mathbf{st}\ \{i \mapsto 3, s \mapsto 3\} \bullet x \coloneqq s \rightarrow SKIP) \qquad \textit{Rules}\ (22), (23), (25) \\
\xrightarrow{x=3} & (\mathbf{st}\ \{i \mapsto 3, s \mapsto 3\} \bullet SKIP) \qquad \textit{Rules}\ (22), (26)
\end{array}$$

It is a series of unobservable steps (in $\top$) until the final observable transition which updates x to 3. No more transitions are possible. (We have used $\xRightarrow{\top}$ to indicate a sequence of more than one $\xrightarrow{\top}$ transitions.) The steps in $\top$ may be interleaved with other processes operating in parallel by Rule (8). The process avoids internal "divergence" by eventually updating a non-local variable.

4 Combining Synchronisation and State-Based Actions

We have so far given a relatively small and straightforward extension to CSP to allow state-based behaviour. However the language is limited in that one cannot combine guards, updates and events in one atomic action. To this end we generalise guards and updates to *specification commands* (Sect. 4.1), which allow arbitrary relationships between pre- and post- states, and allow specification commands to be paired with events (Sect. 4.2).

4.1 Specification Commands

A specification command is of the form $x\colon [R]$, where x is a set of variables and R is a two-state predicate. This construct is based on Morgan's specification command [9], except he uses x_0 and x for pre- and post-state variables, where we use x and x', respectively. The *frame* of the command is the set x, i.e., $\mathit{frame}(x\colon [R]) = x$, and it is the set of variables which may be modified by the specification command. R defines the relationship between the values of pre- and post variables. Variables in the post-state are primed versions, and only variables in the frame x may appear primed in R. For example, a specification command which increments i is written as $i\colon [i' = i + 1]$. A guard as described in the previous section is a special case of a specification command where x is empty and R (therefore) does not refer to the post-state, that is, $[g]$ is now an abbreviation for $\varnothing\colon [g]$, as in [9]. Similarly, an update $x \coloneqq E$ is an abbreviation for $x\colon [x' = E]$. The syntax of processes is extended to allow a specification command as a prefix, which, given these abbreviations, subsumes prefixing by guards and assignments.

$$P ::= (\mathbf{st}\ \sigma \bullet P) \mid (x\colon [R] \to P) \mid \ldots \tag{31}$$

where '...' includes the operators for CSP from (1). We require $\alpha(x\colon [R] \to P) = \alpha(P)$.

As in the previous extension, we define the set of state-based actions, *SCmd*, to contain all specification commands (and allow guard and update abbreviations to appear in transition labels). In keeping with the guard abbreviation, we define $\top$ as any command $\varnothing\colon [g]$ where $[\![g]\!] = \mathit{true}$. The following rule subsumes Rule (22).

$$(x\colon [R] \to P) \xrightarrow{x\colon [R]} P \tag{32}$$

Given a two-state predicate R and states σ and σ', the substitution of σ in the pre-state of R and σ' in the post-state of R is written $R[\sigma, \sigma']$. For instance, if $\sigma = \{i \mapsto 0\}$ and $\sigma' = \{i \mapsto 1\}$, then $(i' = i + 1)[\sigma, \sigma']$ is $(1 = 0 + 1)$. A specification command $x\colon [R]$ is satisfiable when there exists some state σ and values for the frame variables x such that R holds for the pre-state σ and post-state formed from σ updated with the new values for the variables in x.

$$\mathbf{sat}(x\colon [R]) \mathrel{\widehat{=}} (\exists\, \Sigma \bullet (\exists\, V : (x \to \mathit{Val}) \bullet R[\Sigma, \Sigma \oplus V]))$$

The rule for a specification command transition in a local state is given below.

$$\frac{P \xrightarrow{x\colon [R]} P' \quad y = x \cap \operatorname{dom}\sigma \quad z = x \setminus \operatorname{dom}\sigma \quad V \in (y \to \mathit{Val}) \quad \sigma' = \sigma \oplus V \quad \mathbf{sat}(z\colon [R[\sigma, \sigma']])}{(\mathbf{st}\ \sigma \bullet P) \xrightarrow{z\colon [R[\sigma,\sigma']]} (\mathbf{st}\ \sigma' \bullet P')} \tag{33}$$

A process $(\mathbf{st}\ \sigma \bullet P)$ may take a transition labelled by a specification command $z\colon [R[\sigma, \sigma']]$ under the following conditions. P transitions in specification command $x\colon [R]$ to P'. Set y is the subset of variables of x that are in the local

state σ; z is the remaining variables. Choose some new values for the variables in y and call this state V. Then the new state σ' is the same as σ with variables in y updated according to V. Finally, the specification command $z : [R[\sigma, \sigma']]$ must be satisfiable. Then the conclusion of the rule states that $(\mathbf{st}\ \sigma \bullet P)$ transitions to $(\mathbf{st}\ \sigma' \bullet P')$ with label $z : [R[\sigma, \sigma']]$, i.e., the visible behaviour is an update of non-local variables z such that R holds, after variables in the local state σ are replaced by their local values in the pre- and post-states. If z is empty and $[\![R[\sigma, \sigma']]\!]$, the label is $\varnothing : [\mathit{true}]$, i.e., $\top$.

For example, consider a specification command which swaps the values of two variables, i and j, if both are greater than 0, but blocks otherwise.

$$i, j : [R] \qquad \text{where } R \mathrel{\widehat{=}} i > 0 \land j > 0 \land i' = j \land j' = i$$

Consider the execution of $(i, j : [R] \to Q)$ in a context which maps i to 5 and j to 10. The guard is satisfied since both i and j are non-zero, and the result is to swap their values. The following transition is justified by Rules (33) and (32).

$$(\mathbf{st}\ \{i \mapsto 5, j \mapsto 10\} \bullet i, j : [R] \to Q) \xrightarrow{\top} (\mathbf{st}\ \{i \mapsto 10, j \mapsto 5\} \bullet Q)$$

The instantiations of the rule meta-variables are $y = \{i, j\}, z = \varnothing, V = \{i \mapsto 10, j \mapsto 5\} = \sigma'$. Substituting σ and σ' into R gives true, and hence $R[\sigma, \sigma']$ is trivially satisfiable. No other choice for V would give a valid transition, since the satisfiability constraint would not hold.

All rules in Figs. 1 and 2 remain valid. In particular, Rule (33) specialises to Rule (25) with the following instantiations: $x = y = z = \{\}$; hence $V = \{\}$ and $\sigma' = \sigma$. Rule (33) specialises to Rule (26) for $x \notin \operatorname{dom} \sigma$ by replacing the abbreviation $x \coloneqq E$ by $x : [x' = E]$, with the following instantiations: $y = \varnothing, z = \{x\}$, and hence $V = \varnothing$ and $\sigma' = \sigma$. We assume that E does not contain any primed variables, and therefore $\mathbf{sat}(x : [x' = E[\sigma]])$ holds because it simplifies to $(\exists\, \Sigma \bullet (\exists\, w : \mathit{Val} \bullet w = E[\Sigma]))$ which is true by the one-point law. Rule (33) specialises to Rule (27) for $x \in \operatorname{dom} \sigma$, by replacing the abbreviation $x \coloneqq E$ by $x : [x' = E]$, with the following instantiations: $y = \{x\}, z = \varnothing$, and hence, for some v, $V = \{x \mapsto v\}$ and $\sigma' = \sigma \oplus \{x \mapsto v\}$. We assume that E does not contain any primed variables, and therefore $z : [R[\sigma, \sigma']]$ is $\varnothing : [v = E[\sigma]]$.

4.2 Combining State and Synchronisations

To allow more generality, we now allow each action in the language to be a specification command/event pair, written (c, μ). This pair subsumes both specification command prefix and event prefix. If c is $\top$ we abbreviate (c, μ) to μ, and if μ is τ we abbreviate (c, μ) to c. We allow τ as an abbreviation for $(\top, \tau)$. We require

$$\alpha((c, \mu) \to P) = \alpha(P) \quad \text{where } \mu \in \mathit{Event} \Rightarrow \mu \in \alpha(P)$$

The intuition is that an action (c, a) can synchronise on an event a if the guard for c holds, and will update the variables in the frame according to c. For

instance, to correct the problem with possible refusals of *enq* events associated with process (14), the bottom line can be written as

$$([q \neq \langle\rangle], deq!head(q)) \rightarrow (q \coloneqq tail(q)) \rightarrow Q$$

This ensures that the synchronisation on channel *deq* will occur only when q is nonempty, without precluding the choice to enqueue a value.

For syntactic convenience in specifying a sequence of state-based actions to be executed atomically, we allow action pairs to be composed. Such a composition is well-formed only if at most one of the commands has an event a. That is, two events cannot be composed. We use the symbol '$\circ$' to denote composition of action pairs, and overload it to also denote relational composition of specification commands.

$$(c_1, \tau) \circ (c_2, \tau) = (c_1 \circ c_2, \tau) \tag{34}$$

$$(c_1, a) \circ (c_2, \tau) = (c_1 \circ c_2, a) = (c_1, \tau) \circ (c_2, a) \tag{35}$$

The relational composition of two specification commands, $c_1 \circ c_2$, which have the same frame x, is defined by matching the post-state of c_1 to the pre-state of c_2 via intermediate variables x''.

$$x : [R_1] \circ x : [R_2] = x : [\exists\, x'' \bullet R_1[\frac{x''}{x'}] \wedge R_2[\frac{x''}{x}]] \tag{36}$$

The expression $R_1[\frac{x''}{x'}]$ is R_1 with a syntactic replacement of variables x' with x''. Note that this is a different type of substitution to that involving states. For the purposes of defining relational composition when the frames do not match, their frames may be widened according to the rule below.

$$x : [R] = x \cup y : [R \wedge y' = y] \qquad \text{for } x \cap y = \varnothing \tag{37}$$

For example, the dequeuing of an element can be merged with the test of non-emptiness and an event on channel *deq* into a single atomic action as follows.

$$\begin{aligned}
&([q \neq \langle\rangle], deq!head(q)) \circ (q \coloneqq tail(q), \tau) \\
&= ([q \neq \langle\rangle] \circ q \coloneqq tail(q), deq!head(q)) && \text{from (35)} \\
&= (q : [q \neq \langle\rangle \wedge q' = q] \circ q : [q' = tail(q)], deq!head(q)) && \text{from (37)} \\
&= (q : [\exists\, q'' \bullet q \neq \langle\rangle \wedge q'' = q \wedge q' = tail(q'')], deq!head(q)) && \text{from (36)} \\
&= (q : [q \neq \langle\rangle \wedge q' = tail(q)], deq!head(q)) && \text{one-point rule}
\end{aligned}$$

In addition to allowing paired actions, we now require transition labels to be pairs of commands and events. As above, a transition with command c and event μ is written $P \xrightarrow{c,\mu} P'$. Both fields are mandatory for every transition, however, if the command is $\top$ or the event is τ, we omit them, with the minimum label being τ. As an example of a non-trivial label, by the above calculation:

$$[q \neq \langle\rangle] \circ deq!head(q) \circ q \coloneqq tail(q) \rightarrow P \xrightarrow{q : [q \neq \langle\rangle \wedge q' = tail(q)], deq!head(q)} P$$

$$((c, \mu) \to P) \xrightarrow{c,\mu} P \tag{38}$$

$$\frac{P \xrightarrow{c,a} P' \quad a \notin \alpha(Q)}{P \parallel Q \xrightarrow{c,a} P' \parallel Q} \tag{39}$$

and similarly for Q.

$$\frac{P \xrightarrow{c,\mu} P' \quad \mu \neq \checkmark}{P \,;\, Q \xrightarrow{c,\mu} P' \,;\, Q} \qquad \frac{P \xrightarrow{\checkmark} P'}{P \,;\, Q \xrightarrow{\tau} Q} \tag{40}$$

$$\frac{P \xrightarrow{c,a} P' \quad a \in A}{P \backslash A \xrightarrow{c,\tau} P' \backslash A} \qquad \frac{P \xrightarrow{c,\mu} P' \quad \mu \notin A}{P \backslash A \xrightarrow{c,\mu} P' \backslash A} \tag{41}$$

$$\frac{P \xrightarrow{\tau} P'}{P \,[\![\, Q \xrightarrow{\tau} P' \,[\![\, Q} \qquad \frac{P \xrightarrow{c,\mu} P' \quad (c, \mu) \neq (\top, \tau)}{P \,[\![\, Q \xrightarrow{c,\mu} P'} \tag{42}$$

and similarly for Q.

$$\frac{P \xrightarrow{x_1 : [R_1], a} P' \quad Q \xrightarrow{x_2 : [R_2], a} Q' \quad x_1 \cap x_2 = \varnothing}{P \parallel Q \xrightarrow{(x_1 \cup x_2) : [R_1 \wedge R_2], a} P' \parallel Q'} \tag{43}$$

$$\frac{\begin{array}{c} P \xrightarrow{x : [R], \mu} P' \quad y = x \cap \operatorname{dom} \sigma \quad z = x \setminus \operatorname{dom} \sigma \quad V \in (y \to \mathit{Val}) \\ \sigma' = \sigma \oplus V \quad \mathbf{sat}(z : [R[\sigma, \sigma']]) \end{array}}{(\mathbf{st}\ \sigma \bullet P) \xrightarrow{z : [R[\sigma, \sigma']], \mu[\sigma]} (\mathbf{st}\ \sigma' \bullet P')} \tag{44}$$

Fig. 3. Rules for specification command/event pairs

We must now define the rules for paired specification command/event transition labels. This is just a matter of combining the rules from Figs. 1 and 2; the result is given in Fig. 3. Rules (6) and (9) do not change. Rule (44) is a combination of Rules (24) and (33). Note that when c is $\top$, the rules collapse to those for CSP given in Fig. 1, and when μ is τ and c is a guard or update, the rules collapse to those in Fig. 2. Rule (43) allows two specification commands to be conjoined if their frames are disjoint. Recall that to be well-formed a specification command cannot alter variables outside its frame. This allows concurrent (and atomic) updates of distinct variables in separate threads.

4.3 Example

Recall the processes S, R and Sys from (4) and Qu from (14). So that CSP_σ may be compared more readily with CSP, these processes have been defined using communication over channels. We now rewrite these processes using state variables to communicate data. We assume that Sys interacts with its environment by receiving information on channel ci and passing information on co. Firstly, we assume x is the only non-local variable that process $\mathit{Produce}$ in S visibly alters, and therefore $\mathit{Produce}$ does not need to be parameterised. An implementation of $\mathit{Produce}$ may be a process which reads some value from the environment on channel ci then finds the sum to that number, i.e.,

$$\mathit{Produce} \mathrel{\widehat{=}} ci?X \to \mathit{Sum}(X)$$

where $\mathit{Sum}(X)$ is defined in (15). Process $\mathit{Consume}(y)$ used in R in (4) may perform some actions that manipulate non-local variable y (possibly involving

local variables), and then output the result to the environment on channel co. For the purposes of this example we define *Consume* minimally as a process which outputs y on channel co, i.e., $Consume(y) \mathrel{\widehat{=}} co!y \rightarrow SKIP$.

The next transformation we make is to replace communication with the Qu thread by direct operations on the variable q. In addition, we introduce the scope of x and y to contain S and R, respectively. These states are external to S and R because these processes are recursively defined, and would otherwise result in a series of redundant nested states.

$$\begin{array}{l} S \mathrel{\widehat{=}} Produce\,;(q \coloneqq q \frown \langle x \rangle \rightarrow S) \\ R \mathrel{\widehat{=}} y, q : [q = \langle y \rangle \frown q'] \rightarrow (Consume(y)\,;R) \\ Sys \mathrel{\widehat{=}} (\mathbf{st}\ \{q \mapsto \langle\rangle\} \bullet (\mathbf{st}\ \{x \mapsto 0\} \bullet S) \parallel (\mathbf{st}\ \{y \mapsto 0\} \bullet R)) \end{array} \tag{45}$$

Process S involves three layers of state. Within *Produce* are the local variables of *Sum*, i and s, which are not visible. Variable x is external to *Sum* and *Produce* but local to S; while q is external to S but local to *Sys*. The variables i and s are *local* variables, while q is a *shared* variable, however, the semantics makes no distinction.

A trace of *Sys* will appear as a sequence of communications on channels ci and co, interspersed with steps in τ. We build such a trace below. For space reasons we write a process $(\mathbf{st}\ \{x \mapsto 0\} \bullet P)$ as $P^{x=0}$, and write $P \overset{s}{\Longrightarrow} Q$ to represent a transition formed from multiple steps, in which the only externally visible transitions are those in s. Trace (46) represents possible executions of processes S and R. Process S receives the value 2 on channel ci from the environment, the sum to 2 is evaluated and stored in x, and q is updated to contain the value in x. Process R takes y from the queue and outputs it on channel co. Trace (47) is a trace formed from S and R in parallel, observed externally to the local variables x and y. Note that the values of those variables change within the state, and that references to them are replaced in the transition labels. Trace (48) represents a trace of the process *Sys* viewed externally; only communication with the environment is visible, with the updates to q reflected in the local state.

$$S \overset{ci.2/x \coloneqq 3/q \coloneqq q \frown \langle x \rangle}{\Longrightarrow} S \qquad R \overset{y,q:\,[q = \langle y \rangle \frown q']/co.y}{\longrightarrow} R \tag{46}$$

$$(S^{x=0} \parallel R^{y=0}) \overset{ci.2/q \coloneqq q \frown \langle 3 \rangle}{\Longrightarrow} (S^{x=3} \parallel R^{y=0}) \overset{q:\,[q \neq \langle\rangle \wedge q' = tail(q)]/co.3}{\Longrightarrow} (S^{x=3} \parallel R^{y=3}) \tag{47}$$

$$(S^{x=0} \parallel R^{y=0})^{q=\langle\rangle} \overset{ci.2}{\Longrightarrow} (S^{x=3} \parallel R^{y=0})^{q=\langle 3 \rangle} \overset{co.3}{\Longrightarrow} (S^{x=3} \parallel R^{y=3})^{q=\langle\rangle} \tag{48}$$

5 Related Work

5.1 Comparison with CSP

We first note that $(P \parallel P) = P$ does not hold if P accesses non-local variables, even in the absence of internal choice (demonic nondeterminism). Intuitively, this is as expected because successive updates of the same variable could result

in a different final value. Technically, this is because specification commands are interleaved by Rule (39) (and an *SCmd* does not appear in the alphabet of any process), unless accompanied by a synchronisation event (Rule (43)).

The language extensions and semantics we have introduced in this paper provide a notational convenience for specifying state-based process behaviour. However, any process written in CSP_σ can be transformed into a process in CSP if state values are kept as parameters to processes and channels are used to exchange information instead of shared variables. To transform a process in CSP_σ to CSP, any accesses of a variable that is shared among multiple parallel processes must become encapsulated by a separate process. For instance, the queue is represented by a variable in (45), while in (4) it must be represented by a separate process, in this case, $Qu(\langle\rangle)$.

The CSP process in (2) gives an encapsulated data type Qu, which in some contexts is desirable, but can become cumbersome in others. Consider an extension of the CSP process *Sys* (4) that contains n instances of process R. We add a parameter to R, indicated by a subscript $i \in 1..n$, and must distinguish the *deq* channels so that pairs R_i and R_j do not synchronise with each other, but only with Qu. We also parameterise *Consume* in a similar manner, so that interactions with the environment do not need to synchronise between R_is. The $Qu(q)$ process must also be updated to listen on multiple channels. The relevant definitions are given below.

$$\begin{aligned} R_i &\mathrel{\widehat=} deq_i?y \rightarrow (Consume_i(y)\,;\, R_i) \\ Qu(\langle y\rangle \frown q) &\mathrel{\widehat=} \begin{array}{l} enq?x \rightarrow Qu(\langle y\rangle \frown q \frown \langle x\rangle) \\ [\!] \; ([\!]_i\, deq_i!y \rightarrow Qu(q)) \end{array} \end{aligned} \tag{49}$$

We have used a generalised external choice to specify that the *deq* channel can output (synchronise) for any $i \in 1..n$.

Consider a further modification such that each R_i consumes two *successive* elements of the queue. However $R_i \mathrel{\widehat=} deq_i?y \rightarrow deq_i?z \rightarrow (Consume_i(y,z)\,;R_i)$. is insufficient as it does not ensure y and z were successively enqueued, since other deq_j events may interleave between. One solution is to define another event, e.g., $deq2_i?(y,z)$ which will atomically dequeue two elements. The definition is trivial, but requires a third process definition to be added to (49) (we assume that capability to dequeue a single element is still required). The *Sys* process is also updated to hide the new event and include n instances of R, for which we assume a generalised parallel composition operator.

$$\begin{aligned} &Queue2(\langle\rangle) \mathrel{\widehat=} \ldots \qquad Queue2(\langle y\rangle) \mathrel{\widehat=} \ldots \\ &\qquad\qquad\qquad\qquad\quad enq?x \rightarrow \ldots \\ &Queue2(\langle y,z\rangle \frown q) \mathrel{\widehat=} [\!] \; ([\!]_i\, deq_i!y \rightarrow Queue2(\langle z\rangle \frown q)) \\ &\qquad\qquad\qquad\qquad\quad [\!] \; ([\!]_i\, deq2_i!(y,z) \rightarrow Queue2(q)) \\ &S \mathrel{\widehat=} (Produce(x)\,;\,(enq!x \rightarrow S)) \\ &R_i \mathrel{\widehat=} deq2_i?(y,z) \rightarrow (Consume_i(y,z)\,;\,R_i) \\ &Sys \mathrel{\widehat=} (S \parallel (\|_i\, R_i) \parallel Queue2) \backslash (enq.Val \cup (\textstyle\bigcup_i\, deq2_i.V)) \end{aligned} \tag{50}$$

The approach of adding new operations does not scale for complex data types which require arbitrary, atomic combinations of operations. It is more convenient to specify this behaviour directly in a state-based style. Recall from (45) that there is no need for a queue process, as the queue operations may be merged into S and R directly. Similarly, we are able to define the operation of removing two adjacent elements in one atomic action. The *Sys* process, with n instances of R and double-dequeue is shown below.

$$\begin{aligned} S &\mathrel{\widehat=} \mathit{Produce} \,;\, ((q \coloneqq q \frown \langle x \rangle) \rightarrow S) \\ R_i &\mathrel{\widehat=} y, z, q \colon [q = \langle y, z \rangle \frown q'] \rightarrow (\mathit{Consume}_i(y, z) \,;\, R_i) \\ \mathit{Sys} &\mathrel{\widehat=} (\mathbf{st}\ \{q \mapsto \langle\rangle\} \bullet S^{x=0} \parallel (\|_i\ {R_i}^{y=0, z=0})) \end{aligned} \tag{51}$$

Note that there are n declarations of the local variables y and z, but we do not need to subscript them to distinguish the different instances. All references to y or z in R_j will properly reference the correct version.

Given that a new process *Queue*2 does not need to be defined and internal messages do not need to be hidden, (51) is a more compact specification than that given by (50). It is also flexible enough if other operations on the queue are required. In CSP, all shared data must have an opaque type, however, as demonstrated above, in CSP_σ shared data may have their types exposed. Of course, it is a separate question as to whether process algebras such as CSP should be used for defining programs like *Sys*. Hoare gives laws for assignment and state variables, but concludes that the laws are not mathematically convenient, and that "... there are adequate grounds for introducing the assignable program variable as a new primitive concept" [7, Sect. 5.5.3]. We intend CSP_σ to be in keeping with the spirit of this statement, and the specification style of CSP in general.

5.2 Other Work

CSP has been integrated with state-based languages, for instance, with Z by Woodcock & Cavalcanti (Circus) [14], with Object-Z by Smith and Fischer & Wehrheim (CSP-OZ) [13,5], with Action Systems by Butler [2], and with B by Butler & Leuschel [1]. In comparison with these approaches to combining state-based specification with CSP, we have taken a "lightweight" approach, integrating only a single construct for defining state manipulation, and with little change to the underlying syntax and semantics of CSP. Of course, the addition of state tests and updates does not provide the same richness of specification as afforded by a full incorporation of Z, etc., but may provide a useful stepping stone between event- and state-based specifications.

Plotkin's seminal paper on operational semantics [10] defines transition rules for imperative languages with state. There are also many other examples of such semantics in the literature, in particular, the semantics of Hoare and He Jifeng [8], and the semantics for the programming language Occam [6]. Our approach is different in that the state is treated as part of the process, and guards and updates are treated as labels to the transition relation. This allows state accesses to be (perhaps partially) hidden by an outer context which defines the values

of the local state. The traditional operational semantics approach defines the transition relation on program/state pairs, and the state is updated in the rule for each construct (e.g., update). This approach does not so easily support the hierarchical construction of the state as in our approach, with local variables in the traditional style being captured as global variables with syntactic restrictions. In the approach adopted here, by treating state access as transition labels, the state-based reasoning is 'quarantined' to a single, general rule (Rule (44)), allowing the construct rules, e.g., Rule (22) and Rule (32), to be defined concisely, and without explicit reference to a particular state. This extends to more complex constructs, for instance, the transition rules for *conditional* can be given without reference to state (see Rule (52)).

$$(\textbf{if } b \textbf{ then } P \textbf{ else } Q) \xrightarrow{[b]} P \qquad (\textbf{if } b \textbf{ then } P \textbf{ else } Q) \xrightarrow{[\neg b]} Q \tag{52}$$

6 Conclusions

In this paper we have given an extension to the CSP language which allows state-based constructs to be integrated with inter-process synchronisation and other CSP constructs. The extension is given an operational semantics, defined so that it is also an extension of the CSP operational semantics: all existing transitions are preserved. The approach taken to defining the transition rules is novel in that the state is maintained as part of the process, instead of a meta-level construct in the rules, and that, therefore, transitions are labelled by specification commands. This enables a more compact presentation and naturally leads to hierarchical definition of states.

The work was motivated when developing a semantics for Behavior Trees [3], a notation used for capturing natural language requirements of large systems. Such requirement documents often mix styles and levels of specification, e.g., a system of systems will specify both the interactions and the internal operations of the processes involved. In future work we will extend CSP_σ to handle the full range of Behavior Tree constructs. In preparation for developing code from CSP_σ, we will define a trace-based semantics for refinement of CSP_σ.

Because the extension builds on existing constructs and semantics, the state-based rules should fit with existing tool support for CSP, such as FDR [4]. However, state-space explosion will become an issue with unrestricted types, and evaluation strategies must be devised to avoid efficiency issues with checking satisfiability in Rule (44).

Acknowledgments. The authors thank three anonymous reviewers for helpful comments on this paper. Ian Hayes would like to acknowledge his Visiting Professorship at, and the hospitality of, the University of Newcastle (UK), and support by the EPSRC-funded Trustworthy Ambient Systems (TrAmS) Platform Project.

References

1. Butler, M.J., Leuschel, M.: Combining CSP and B for Specification and Property Verification. In: Fitzgerald, J.S., Hayes, I.J., Tarlecki, A. (eds.) FM 2005. LNCS, vol. 3582, pp. 221–236. Springer, Heidelberg (2005)
2. Butler, M.J.: A CSP Approach to Action Systems. PhD thesis, Computing Laboratory, Oxford Univ. (1992)
3. Dromey, R.G.: Formalizing the transition from requirements to design. In: Jifeng, H., Liu, Z. (eds.) Mathematical Frameworks for Component Software: Models for Analysis and Synthesis, River Edge, NJ, USA. Component-Based Development, pp. 156–187. World Scientific Publishing Co., Inc., Singapore (2006)
4. FDR2 user manual (2005), `http://www.fsel.com/fdr2_manual.html`
5. Fischer, C., Wehrheim, H.: Model-Checking CSP-OZ Specifications with FDR.. In: Araki, K., Galloway, A., Taguchi, K. (eds.) Integrated Formal Methods, 1st International Conference, Proceedings, pp. 315–334. Springer, Heidelberg (1999)
6. Gurevich, Y., Moss, L.S.: Algebraic operational semantics and Occam. In: Börger, E., Kleine Büning, H., Richter, M.M. (eds.) CSL 1989. LNCS, vol. 440, pp. 176–192. Springer, Heidelberg (1990)
7. Hoare, C.A.R.: Communicating Sequential Processes. Prentice-Hall, Inc., Englewood Cliffs (1985)
8. Hoare, C.A.R., Jifeng, H.: Unifying Theories of Programming. Prentice-Hall, Englewood Cliffs (1998)
9. Morgan, C.: Programming from Specifications, 2nd edn. Prentice-Hall, Englewood Cliffs (1994)
10. Plotkin, G.D.: A structural approach to operational semantics. J. Log. Algebr. Program. 60-61, 17–139 (2004)
11. Roscoe, A.W.: The Theory and Practice of Concurrency. Prentice-Hall, Englewood Cliffs (1998)
12. Schneider, S.: Concurrent and Real-time Systems: The CSP Approach. Wiley, Chichester (2000)
13. Smith, G.: A semantic integration of Object-Z and CSP for the specification of concurrent systems. In: Fitzgerald, J.S., Jones, C.B., Lucas, P. (eds.) FME 1997. LNCS, vol. 1313, pp. 62–81. Springer, Heidelberg (1997)
14. Woodcock, J.C.P., Cavalcanti, A.L.C.: The semantics of circus. In: Bert, D., Bowen, J.P., Henson, M.C., Robinson, K. (eds.) B 2002 and ZB 2002. LNCS, vol. 2272, pp. 184–203. Springer, Heidelberg (2002)